Titre: Title:	Génération automatique de données de test visant la couverture des branches de logiciels écrits en langage C/C
Auteur: Author:	Sébastien Lapierre
Date:	1998
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Lapierre, S. (1998). Génération automatique de données de test visant la couverture des branches de logiciels écrits en langage C/C [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/6853/

UNIVERSITÉ DE MONTRÉAL

GÉNÉRATION AUTOMATIQUE DE DONNÉES DE TEST VISANT LA COUVERTURE DES BRANCHES DE LOGICIELS ÉCRITS EN LANGAGE C/C++

SÉBASTIEN LAPIERRE
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE ET DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
MAI 1998

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

GÉNÉRATION AUTOMATIQUE DE DONNÉES DE TEST VISANT LA COUVERTURE DES BRANCHES DE LOGICIELS ÉCRITS EN LANGAGE C/C++

présenté par: LAPIERRE Sébastien

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

M. DAGENAIS Michel, Ph.D., président

M. MERLO Ettore, Ph.D., membre et directeur de recherche

M. BOUDREAULT Yves, M.Sc., membre

À tous mes proches, Dominique et Charles, Ewa, François et Lisette, pour m´avoir soutenu et encouragé pendant toutes ces années.

Remerciements

Je voudrais remercier Ettore Merlo, mon directeur, pour son aide essentielle, pour ses encouragements tout au long de ce travail de maîtrise ainsi que pour son support financier.

Je voudrais également souligner la participation importante de Giulio Antoniol de l'*Istituto per la Ricerca Scientifica e Technologica* d'Italie ainsi que le support financier apporté par le Conseil de Recherche en Sciences Naturelles et en Génie du Canada (CRSNG).

Je voudrais de plus remercier Yves Boudreault, Richard Prégent ainsi que toute l'équipe du cours Informatique 3.307 pour m'avoir donné la chance d'enseigner et pour m'avoir guidé et soutenu lors de cette expérience des plus passionnante et enrichissante.

Résumé

La phase de tests de logiciels est une activité d´extrême importance lors d´un effort de développement ou de maintenance d'un programme. Afin d´arriver à un niveau satisfaisant de confiance dans le logiciel testé, des techniques de tests à boîte blanche sont habituellement préconisées. Cependant, la détermination des données de tests nécessaires, données à utiliser en entrée lors des tests effectués, représente une difficulté majeure et n´est actuellement accomplie que manuellement ou en utilisant des techniques fondées sur les probabilités ou le hasard.

Le présent projet s´intéresse à la génération automatique de données de tests capables d´assurer la couverture des branches logiques contenues dans un logiciel testé. Pour arriver à ce but, la structure interne d´un logiciel est tout d´abord représentée à l´aide d´un graphe appelé CFG. À partir de ce graphe, les arcs non-contraints du programme sont déterminés. Ces arcs permettent d´identifier le sous-ensemble minimal d´arcs qu´il sera nécessaire d´exercer afin d´obtenir une couverture totale des arcs du programme.

Pour générer des données de tests couvrant un arc non-contraint e du CFG, associé

à une branche donnée du programme analysé, la technique employée fait usage d´un arbre d´exécution partiel représentant les chemins d´exécution possibles, se terminant à l´arc e et étant limités par un seuil maximal du nombre d´itérations des instructions du programme.

Une exécution symbolique est accomplie sur chaque chemin de l´arbre d´exécution partiel considéré. Les techniques traditionnelles d´exécution symbolique ont été étendues pour pouvoir traiter les manipulations et l´arithmétique de pointeurs, les pointeurs à fonction ainsi que certaines particularités relevant des langages orientés-objets. Le résultat de l´exécution symbolique appliquée à l´arbre d´exécution partiel est une expression logique représentant une subjonction entre les conditions de chemins, ou conditions de parcours, caractérisant chaque chemin analysé.

La validation de cette expression, effectuée par une optimisation de l´expression logique transformée en problème d´optimisation linéaire, permettra de déterminer les valeurs d´entrées à utiliser afin de forcer une exécution quelconque couvrant l´arc e.

L´avantage de procéder à un traitement d´une collection de chemins d´exécution est qu´il est ainsi possible de contourner la majeure partie des problèmes causés par les chemins infaisables ainsi que par les chemins non-linéaires dans les programmes.

Un prototype à été réalisé et des expériences ont été effectuées sur des programmes écrits en C et en C++. Les résultats très prometteurs obtenus indiquent que le traitement des arcs non-contraints nous permet de réduire d´environ 80 % le nombre d´arcs à considérer tout en atteignant une couverture d´environ 99 % des branches

du programme pour les programmes analysés.

the software, with the particularity that each of these paths will end at edge e and will be limited by a given maximum number of iterations of the software statements.

Symbolic execution is performed on the execution paths composing the partial execution tree. Traditional symbolic execution techniques have been extended in order to treat pointer manipulations and arithmetic, function pointers and a number of object-oriented features. Resulting from the symbolic execution phase is a logical expression representing a disjunction between each path condition associated with the individual paths within the partial execution tree.

Transforming this expression into a linear optimization problem and solving this problem yields input data to be used while testing. This data will force an execution passing through any one of the paths composing the analyzed partial execution tree and covering goal edge e.

The distinct advantage of reasoning on an execution tree is that most problems associated with infeasible paths and non-linear paths within software programs are avoided.

A prototype has been created and experiments have been performed on C and C++ programs. Very promising results indicate that a reduction of approximately 80% can be obtained in the number of edges to be treated while considering only the unconstrained edges. Using this reduced edges set, input data was generated that achieved 99% branch coverage for the programs in our testbed.

Liste des figures

Introduction

En génie logiciel, la phase de tests d´un logiciel consiste à tenter de trouver des erreurs présentes dans un logiciel donné, ou de déclarer ce dernier valide si aucune erreur n´a pu être détectée. Pour arriver à cette fin, des techniques de tests à boîte noire et de tests à boîte blanche sont souvent utilisées. Selon les techniques de tests à boîte noire, un logiciel est exécuté et des valeurs quelconques d´entrées lui sont fournies. Ces valeurs d´entrées sont appelées jeux de tests ou données de tests pour le logiciel testé. Une fois l´exécution terminée avec ces données, une vérification de conformité est effectuée entre les résultats d´exécution obtenus et les résultats d´exécution anticipés, habituellement déduits selon les spécifications du logiciel en fonction des données de tests utilisées. Lorsque les résultats obtenus sont conformes aux résultats anticipés, le logiciel est considéré valide selon le jeux de tests utilisé. Par contre, une non-conformité entre les résultats obtenus et les résultats anticipés indique qu´une erreur potentielle vient d´être détectée dans le logiciel testé.

Une lacune liée aux tests basés strictement sur des techniques de tests à boîte noire est qu'ils ne sont aucunement liés à la structure interne du logiciel. Ainsi, les

données de tests utilisées seront par exemple générées au hasard ou sur la base d'une étude statistique des conditions d´utilisation réelles du logiciel testé. Avec de telles données, une particularité dans la structure du logiciel testé ne sera ni détectée ni particulièrement sollicitée. Certaines erreurs intimement liées à la structure du logiciel pourront donc demeurer introuvées ou demander un nombre très grand de tests avant d'être détectées. De plus, aucune mesure structurelle d´exhaustivité des tests accomplis ne pourra être déterminée, ce qui permettrait de savoir à quel point tous les aspects structurels du logiciel ont été exercés par les tests effectués.

Les tests à boîte blanche, ou boîte de verre, constituent une autre approche aux tests de logiciels qui est beaucoup plus propice à l´établissement d´une mesure d´exhaustivité des tests accomplis. En effet, les tests à boîte blanche se basent sur une analyse de la structure interne d´un logiciel et tentent d´établir un certain niveau de couverture de cette structure. La couverture d´un logiciel consiste à déterminer ou à utiliser des données de tests qui permettront d´exercer le plus exhaustivement possible un certain aspect du logiciel. Par exemple, une phase de tests basée sur ce principe tentera d´obtenir la couverture de tous les énoncés présents dans le logiciel testé, ou de toutes les branches logiques contenues, ou de critères basés sur l'analyse de flux de données tels que la couverture des chaînes définition-utilisation.

Il est considéré que l'obtention d´une couverture d´environ 80% des branches d´un logiciel, lors de la phase de tests, représente un très bon indice de confiance en sa validité.

Le présent projet s´intéresse à la génération, de manière automatique, de données de tests qui permettront d´obtenir un haut degré de couverture des branches du logiciel testé. Les données générées automatiquement par notre outil pourront par la suite être utilisées durant une phase de tests à boîte noire du logiciel, afin de vérifier que son comportement soit adéquat. Pour un logiciel ainsi testé, la couverture d´un fort pourcentage, voir de la majorité de ses branches, induira un très haut niveau de confiance en sa qualité et sa validité.

La génération automatique de jeux de tests représente un intérêt scientifique et industriel certain puisque, à l´heure actuelle, aucune méthode complète ni aucun outil ne permet de fournir de tels jeux de tests. Plusieurs recherches portant sur le sujet ont été conduites par le passé, mais aucune de ces études n´a mené à l´obtention d´un outil pleinement opérationnel. Un aperçu des méthodes développées ainsi que des références aux recherches passées portant sur la génération automatique de jeux de tests vous seront proposés à la section 1.1. Par la suite, la méthode particulière que nous avons développée sera explicitée à la section 1.2, qui sera suivie par une présentation des résultats expérimentaux obtenus avec l´outil bâtit pour implanter cette méthode à la section 1.3. Le présent mémoire se terminera avec une discussion des résultats présentés, à la section 1.4, suivie par une présentation, à la section 1.5, des conclusions qui peuvent être tirées du travail accompli, puis par une présentation des recherches futures envisageables à la suite de ce projet, à la section 1.6. Veuillez noter que trois articles ont été annexés au présent ouvrage ; ils contiennent l´essentiel

des informations concernant les travaux effectués et seront utilisés comme références lors des explications sommaires données dans ce mémoire.

Chapitre 1

Génération automatique de données de tests pour logiciels

1.1 Méthodes générales employées pour la génération automatique de données de tests

Plusieurs équipes de chercheurs ont établi ou tenté d´améliorer, au fil des années, les techniques de génération automatique de jeux de tests pour logiciels. De manière générale, deux approches distinctes peuvent être identifiées, soit la génération automatique basée sur une approche dynamique et la génération automatique basée sur l´exécution symbolique, une approche hybride entre les analyses classiques de natures statique et dynamique.

Il est à noter que ces efforts de recherche ont strictement porté sur des programmes

écrits en des langages procéduraux et qu´aucune recherche n´à été effectuée pour étendre ces techniques de génération vers des programmes écrits en des langages orientés-objets.

1.1.1 Génération basée sur une approche dynamique

Lors d´une génération de données de tests basée sur une approche dynamique, le logiciel à tester est littéralement exécuté mais le déroulement de son exécution est soumis à un contrôle externe. Ce module de contrôle externe sera à même d´analyser le chemin d´exécution emprunté dans le logiciel testé et pourra déterminer si ce chemin correspond bel et bien à un chemin cible déterminé pour lequel des données de tests doivent être générées. Lorsque l´exécution réelle s´écarte du chemin d´exécution ciblé, une analyse dynamique du point de bifurcation du flux de contrôle sera effectuée, laquelle tentera de déterminer les valeurs qu´auraient dû avoir les données d´entrées fournies afin d´assurer une exécution suivant effectivement le chemin d´exécution ciblé. L´exécution sera poursuivie de la sorte jusqu´à l´atteinte de la fin du chemin ciblé, lorsque possible, et les données d´entrées ainsi créées correspondront aux données de tests recherchées. Une description quelque peu plus complète et des références aux travaux concernés peuvent être consultés à la section 2.1 de l´article présenté à l´annexe A.

1.1.2 Génération basée sur l´exécution symbolique

L´exécution symbolique est en quelque sorte une simulation d´exécution plutôt qu´une exécution réelle d´un logiciel. Lors de cette opération, les instructions composant le logiciel sont simulées en séquence, suivant l´ordonnancement indiqué par un chemin d´exécution déterminé. La simulation de l´exécution d´une instruction compose en fait le coeur d´un module d´exécution symbolique et est définie pour chaque type d´instruction disponible dans un langage donné. Lors de l´exécution symbolique, des variables hypothétiques représentant les variables réelles du programme sont créées et manipulées en mémoire de l´exécuteur, ce qui permet d´actualiser en cours d´exécution l´état de chaque variable utilisée.

Une caractéristique propre à l´exécution symbolique repose sur l´introduction de symboles dans les valeurs attribuées aux variables du programme. Ces symboles, données inconnues lors de l´exécution symbolique, proviennent en fait d´opérations d´entrée effectuées en cours d´exécution. L´exécution d´une opération d´entrée étant simulée, la valeur réellement obtenue en entrée est considérée inconnue et est représentée par un symbole unique dans l´exécuteur. Les opérations manipulant des valeurs obtenues en entrée manipuleront donc des symboles et bâtiront de fait des expressions symboliques à partir de ces symboles.

La génération de données de tests basée sur l´exécution symbolique repose donc sur une simulation d´exécution telle que décrite précédemment. Cependant, sachant que cette exécution est conduite le long d´un chemin d´exécution donné, les valeurs

des conditions composant les instructions de transfert de contrôle du programme qui sont rencontrées le long de ce chemin sont accumulées dans une structure appelée *pc*, acronyme signifiant "path condition" et pouvant être traduit par "condition de chemin". Au terme de l'exécution symbolique le long d'un chemin donné, le *pc* contiendra donc des expressions symboliques qui, liées par des conjonctions, formeront une seule expression symbolique représentant la globalité des conditions du chemin et devant être validée pour assurer que l'exécution se déroule bel et bien le long du chemin considéré. La section 1.2.2 contient un exemple simple qui aide à comprendre les concepts présentés.

La dernière étape de la génération consiste à tenter de déterminer les valeurs à donner aux symboles du *pc* afin de le valider. Les valeurs ainsi trouvées correspondront aux données d'entrée à fournir éventuellement au programme, lors de son exécution réelle, pour assurer que le chemin d'exécution suivi corresponde au chemin considéré lors de la phase d'exécution symbolique.

Les références complètes aux travaux portant sur l'utilisation d'exécution symbolique en génération de données de tests sont disponibles à la section 2.2 de l'article présenté à l'annexe A.

1.2 Description de notre approche

L'approche que nous avons développée nous permet de générer automatiquement des données de tests pour des logiciels écrits en langages C ou C++, suivant donc des

paradigmes procéduraux ou orientés-objets. Pour arriver à ce but, diverses avenues ont été exploitées et combinées. Parmi ces avenues, notons principalement

- l´utilisation d´arbres d´exécution ;
- l´exécution symbolique ;
- les arcs non-contraints d´un programme ;
- le traitement des pointeurs ;
- la représentation de hiérarchies de classes dans un système orienté-objets.

Les sous-sections suivantes présentent l´essentiel des notions concernant les points précédemment exposés, dans un ordre qui rendra leur compréhension plus accessible. Notez qu´une description beaucoup plus complète des notions abordées est disponible dans les articles annexés. Ainsi, l´article de l´annexe C contient une présentation approfondie des arcs non-contraints ainsi qu´une étude de leur importance en génération de données de tests, tel que vu à la section 1.2.1 de manière sommaire. L´article de l´annexe B, pour sa part, s´intéresse plutôt aux particularités associées à la génération automatique de données de tests pour logiciels orientés-objets, notions résumées à la section 1.2.4. Finalement, l´article situé en annexe A contient l´essentiel de la présentation de notre approche générale pour réaliser la génération automatique. Les notions contenues dans cet article feront l´objet de toutes les autres sous-sections de la section 1.2.

1.2.1 Arcs non-contraints

Lors de la génération automatique de jeux de tests pour un programme P, le programme est tout d´abord analysé lexicalement puis syntaxiquement afin de produire un graphe de flux de contrôle (CFG) représentant sa structure interne. Dans un CFG, un noeud est associé à une instruction de P et un arc correspond à un transfert de contrôle possible entre les instructions correspondantes de P.

Des arcs non-contraints peuvent être identifiés sur le CFG et correspondent à des branches essentielles de P. Un arc non-contraint est défini en fonction des relations de dominance et de post-dominance dans un graphe dirigé ayant un arc d´entrée unique e_{in} et un arc de sortie unique e_{out}.

La relation de dominance entre arcs peut être énoncée comme suit : un arc e_1 du CFG domine un autre arc e_2 du CFG si tous les chemins possibles, dans le CFG, partant de l´arc e_{in} et atteignant l´arc e_2 passent nécessairement par e_1.

De manière similaire, la relation de post-dominance entre arcs correspond à ce qui suit : un arc e_1 du CFG domine un autre arc e_2 du CFG si tous les chemins possibles, dans le CFG, partant de l´arc e_2 et atteignant l´arc e_{out} passent nécessairement par e_1.

Un arc non-contraint est défini comme un arc du CFG qui n´est dominé par aucun autre arc du CFG et qui n´est post-dominé par aucun autre arc du CFG. Soit l´ensemble UE représentant tous les arcs non-contraints du CFG correspondant à P, la définition d´un arc non-contraint implique que tout arc du CFG qui n´est pas

prédicats rencontrés le long du chemin d´exécution sont accumulées dans une structure appelée *pc*.

```
1 :   void main(void) {
2 :       int x, y;
3 :       scanf("%d", &x);
4 :       y = x + 2;
5 :       if (y > 5)
6 :           printf("message 1");
7 :       else
8 :           printf("message 2");
9 :   }
```

FIG. 1.1: *Programme utilisé en exemple.*

Par exemple, soit le programme de la figure 1.1 et soit un chemin d´exécution défini par la suite d´instructions donnée par les numéros de lignes $< 1, 2, 3, 4, 5, 8, 9 >$, alors l´état final des variables sera le suivant :

$$x : s_1,\ y : s_1 + 2$$

s_1 étant un symbole ayant été affecté à x lors de l´exécution symbolique de la ligne 3. Le *pc* bâti correspondrait à l´expression :

$$pc : ((s_1 + 2) > 5) = FAUX$$

car le chemin d´exécution spécifié exige que la branche *else* de la conditionnelle soit empruntée.

L´exécution symbolique est définie pour tous les types d´instructions de base pouvant former un programme en langage C ou C++. Les détails décrivant ces opérations sont disponibles aux annexes suivantes :

- article présenté en annexe A pour l´exécution symbolique de programmes écrits en langage C ;
- article présenté en annexe B pour l´exécution symbolique de programmes écrits en langage C++.

Le traitement des pointeurs, des pointeurs à fonction et de l´allocation dynamique de mémoire ont été réalisés dans l´exécuteur symbolique. La section 1.2.3 présente les principales difficultés mises en jeu par leur manipulation. L´exécution symbolique à de plus été étendue pour pouvoir traiter les aspects d´un langage orienté-objets, soit le C++. Ceci représente une nouveauté car aucun effort de recherche par d´autres équipes de chercheurs n´avait été effectué précédemment en ce sens. La section 1.2.4 sera consacrée à la présentation des modifications nécéssaires à apporter à un exécuteur symbolique conventionnel pour lui permettre de traiter des objets et des classes.

1.2.3 Traitement des pointeurs

Le traitement des pointeurs représente souvent un obstacle aux analyses traditionnelles de nature statique. Cependant, le fait que l´exécution symbolique ne s´intéresse qu´à un chemin d´exécution précis dans le programme constitue une différence importante avec les analyses statiques et permet une manipulation plus aisée des opérations

sur les pointeurs.

En effet, lors d'opérations usuelles sur les pointeurs, telles que l'adressage (&) et la déréférenciation (* ou $\rightarrow$), le pointeur spécifié est unique, connu, et son contenu est lui aussi unique et parfaitement connu à un point donné de l'exécution. Son traitement est donc simple et ne nécessite que l'établissement de liens mémoire entre les variables manipulées par l'exécuteur symbolique afin de représenter les liens mémoire réels caractérisant les pointeurs utilisés dans le programme. Les pointeurs à fonction se basent sur le même principe que les pointeurs usuels et établissent leurs liens vers des CFG en mémoire plutôt que vers des variables.

L'allocation dynamique de mémoire est elle aussi réalisée assez directement en créant de nouveaux éléments mémoire de niveau global (dans le "heap") et en liant ces éléments aux pointeurs auxquels ils sont associés dans le programme.

Les algorithmes complets représentant ces fonctionnalités sont disponibles à la fin de l'article présenté en annexe A.

1.2.4 Exécution symbolique de programmes orientés-objets

L'exécution symbolique de programmes orientés-objets nécessite l'établissement en mémoire de l'exécuteur des informations suivantes :

- l'identification non-ambigüe des classes, objets et méthodes du programme ;
- la hiérarchie définie pour les classes du programme ;
- la définition ainsi que les redéfinitions de méthodes pour les différentes classes ;

- le fait qu´une méthode ait été définie comme virtuelle ou non ;
- l´identification d´attributs statiques de classe.

L´identification des entités du programme repose, dans notre approche, sur une notation de quadruplet où sont spécifiés des indices pour représenter :

- l´identificateur unique de classe ;
- l´identificateur de méthode/fonction, commun à toutes les méthodes virtuelles d´une même hiérarchie de classe ;
- l´identificateur de variable/attribut, unique pour toutes les classes d´une même hiérarchie de classes ;
- le type, correspondant à un identificateur de classe, d´une variable/attribut, information nécessaire lors de l´établissement, durant l´exécution, de la méthode réellement invoquée en présence d´un appel de méthode virtuelle par un pointeur à objet.

La hiérarchie des classes est en fait un ensemble d´arbres dans lesquels les classes de base se retrouvent à la racine. Cette information est importante car, afin de trouver la bonne méthode à invoquer lors d´un appel de méthode virtuelle par l´intermédiaire d´un pointeur à objet, la méthode associée au pointeur sera définie comme appartenant à la classe de base mais un parcours devra être effectué dans l´arbre hiérarchique afin de déterminer la méthode réelle appelée selon la classe de l´objet pointé. Bien entendu, ce cheminement ne sera employé que si la méthode a été identifiée comme étant virtuelle.

à la figure 1.2.

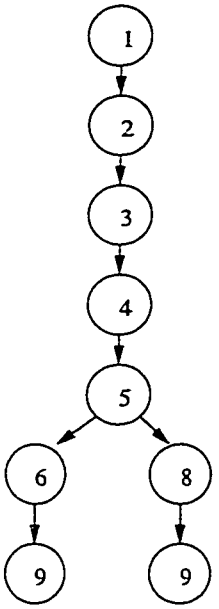


FIG. 1.2: *Arbre d´exécution pour le programme de la figure 1.1.*

Notre approche fait un usage très innovateur du concept d´arbre d´exécution. Afin de générer des données de tests pour couvrir un arc e d´un programme P, l´arbre d´exécution complet ET de P est tout d´abord réduit à un arbre d´exécution partiel ET_p répondant aux deux critères suivants :

- les terminaisonss de ET_p correspondent à l´arc e ;
- les chemins d´exécution composant ET_p contiennent un maximum de n itérations.

L´arbre d´exécution partiel ET_p pour un programme P est tout d´abord généré avec une limite d´itérations $n = 0$ et considérant comme arc cible un arc non-contraint e de P. Chaque chemin d´exécution composant ET_p fait par la suite l´objet d´une

exécution symbolique, ce qui nous donne des *pc* représentant les conditions logiques à remplir pour assurer une exécution le long de chacun de ces chemins.

Puisque les différents chemins d´exécution possibles pour atteindre e ont été unifiés pour former ET_p, il découle que les *pc* générés et associés à ces chemins peuvent subir une unification similaire. En fait, l´unification des *pc* générés revient à combiner tous les *pc* à l´aide de subjonctions logiques. Le résultat de cette unification à été appelé $PBCC$, acronyme signifiant "partial branch coverability condition" et pouvant être traduit par "condition partielle de couverture de branche". L´intérêt du $PBCC$ est que nous obtenons ainsi une seule équation, composée par la subjonction de plusieurs *pc*, qui, si elle est validée, nous assurera qu´un des *pc* qui la composent sera validé. Le fait qu´un *pc* du $PBCC$ soit validé implique qu´un des chemins composant l´arbre d´exécution partiel ET_p est faisable.

Un chemin faisable est défini comme étant un chemin, dans le programme, le long duquel une exécution pourra réellement se propager, par opposition à un chemin infaisable qui ne pourra être exercé par aucune exécution réelle.

Un grand avantage de notre approche est que nous pouvons ainsi contourner la majeure partie des problèmes liés aux chemins infaisables dans un programme.

Prenons un chemin quelconque contenu dans un programme. Étant donné les conditions rencontrées le long de ce chemin, conditions de nature opposées, par exemple, le chemin considéré sera peut-être infaisable, ce qui veut dire qu´aucune exécution réelle ne pourra emprunter ce chemin d´exécution. Il est impossible de déterminer,

de manière générale, si un chemin d´exécution est faisable ou non. L´analyse de chemins individuels, approche traditionnellement adoptée par les équipes de génération de données de tests, se confronte donc à un grave problème d´indécidabilité dû à la présence de chemins infaisables dans les programmes. Notre approche, puisqu´elle traite les chemins non pas individuellement mais plutôt par regroupements, nous permet de contourner ce problème d´indécidabilité à la condition qu´au moins un des chemins composant l´arbre d´exécution partiel soit faisable.

De plus, le fait de considérer les chemins par regroupements permet aussi de contourner certains problèmes dûs aux chemins non-linéaires dans les programmes. Un chemin non-linéaire est un chemin le long duquel les variables impliquées dans les prédicats des transferts de contrôle rencontrés subissent des opérations non-linéaires, par exemples des multiplications entre elles. Une exécution symbolique effectuée le long d'un tel chemin produira en sortie un *pc* dans lequel les symboles constituent une expression de degré supérieur à 1. Les générateurs de données de tests utilisant l´exécution symbolique étant majoritairement basés sur un résoluteur d´équations de nature linéaires, un chemin non-linéaire représente un chemin qui ne peut, dans ce contexte, être traité.

Notre approche offre l'avantage de considérer plusieurs chemins en alternative, ce qui implique que la présence de chemins non-linéaires dans l´arbre d´éxécution partiel sera sans effet en autant qu'un chemin linéaire soit présent en alternative.

1.2.6 Détermination des données de tests

La condition partielle de couverture de branche ($PBCC$) est une expression de nature purement logique (booléenne) comprenant des symboles pour lesquels des valeurs doivent être attribuées. Ces valeurs correspondront aux données de tests à utiliser afin de forcer une exécution le long d'un des chemins de l'arbre d'exécution partiel. Les chemins se terminant tous à un même arc non-contraint e, nous assurerons ainsi une exécution quelconque couvrant l'arc e.

Pour réaliser la détermination proprement dite des données de tests, nous transformons le $PBCC$ logique en un problème d'optimisation linéaire pour lequel la fonction d'optimisation est non-significative. L'optimisateur utilisé, $CPLEX$, nous donnera en sortie les valeurs à attribuer aux symboles, donc les valeurs d'entrées à utiliser, pour couvrir un arc non-contraint e, pourvu qu'un des chemins composant l'arbre d'exécution partiel soit faisable.

Rappelons que le problème soumis à l'optimisateur découle de l'analyse d'un arbre d'exécution partiel pour lequel un seuil maximal du nombre d'itérations des instructions avait été établi. Il est possible que, dû à cette contrainte sur les itérations permises, le problème n'accepte pas de solution. Dans un tel cas, le processus complet de génération de jeux de tests pour couvrir l'arc analysé dans le programme pourra être répété en utilisant un seuil itératif plus élevé que le seuil précédemment employé.

1.3 Résultats expérimentaux obtenus

Un outil de génération automatique de jeux de tests à été créé afin d´implanter les principes développés pendant le projet. À l´aide de cet outil, 17 programmes ont étés analysés. De ces programmes, 12 programmes étaient écrits en C et 5 programmes étaient écrits en C++.

1.3.1 Résultats pour les programmes procéduraux

Pour les programmes écrits en C, qui totalisaient environ 1500 lignes de code source, un calcul des arcs non-contraints à permis de constater que les arcs non-contraints représentaient environ 20% de tous les arcs des programmes de l´expérience.

La génération de données de tests à été effectuée pour ces programmes en considérant, en un premier temps, uniquement les arcs non-contraints des programmes, puis en considérant tous les arcs des programmes. Dans un cas comme dans l´autre, une couverture d´environ 99% des branches des programmes a été accomplie.

De manière générale, les arbres d´exécution partiels bâtis en cours d´analyse étaient comparables, tant pour la génération de données en considérant uniquement les arcs non-contraints des programmes que pour la génération de données en considérant tous les arcs des programmes. Pour arriver à ce résultat, une comparaison des données suivantes à été effectuée pour les deux types de génération :

- le nombre moyen de chemins contenus dans les arbres d´exécution partiels ;
- le nombre moyen d´instructions contenues dans les arbres d´exécution partiels ;

– le nombre de symboles générés par les analyses effectuées.

Une réduction importante de 80% du nombre d´arcs considérés a donc été obtenue. Cette réduction assure cependant un degré de couverture équivalent et n´amène aucune difficulté supplémentaire à la génération de données de tests. De plus, une réduction d´environ 77% du temps nécessaire pour accomplir cette génération à été enregistrée (la génération pour couvrir les arcs non-contraints a nécessité environ 30 secondes de temps de calcul).

Le degré de couverture atteint (99%) a été atteint en considérant comme limite au nombre d´itérations permises les valeurs 0 et 1. Des analyses ont aussi été effectuées en considérant une limite de 2 itérations, ce qui a permis de constater que la taille des arbres d´exécution partiels explosait de manière exponentielle en fonction de la limite du nombre d´itérations permises.

Une présentation plus approfondie des résultats obtenus et présentés dans cette section est disponible dans les articles des annexes A et C.

1.3.2 Résultats pour les programmes orientés-objets

Cinq expériences ont été effectuées sur des programmes écrits en C++..Ces programmes, malgré leurs petites tailles, avaient pour but de démontrer que la démarche de génération de données de tests était applicable aux programmes conçus suivant un paradigme objet et que les traitements proposés permettaient efficacement d´analyser :

mins en considérant non pas les chemins d´exécution individuellement mais plutôt alternativement sous la forme d´un arbre d´exécution ;

- limiter le nombre d´arcs à considérer, et de ce fait le temps de calcul requis pour effectuer la génération, sans pour autant augmenter la difficulté de génération des données de tests.

L´explosion de la taille des arbres d´exécution en fonction de la limite des itérations permises impose cependant une contrainte à la taille des CFG pouvant être traités par la méthode proposée. Une solution à ce problème est d´effectuer des tests unitaires plutôt que des tests d´intégration, limitant ainsi la taille des problèmes à traiter.

1.5 Conclusion

Le présent mémoire avait pour but de présenter une approche permettant d´effectuer la génération automatique de données de tests pour logiciels écrits en langage C et C++. L´approche proposée repose principalement sur l´utilisation d´arcs non-contraints, d´arbres d´exécution et d´exécution symbolique étendue à des concepts du paradigme objet.

Des expériences accomplies sur 17 programmes montrent que la technique développée est extrêmement prometteuse et que le but initial du projet, soit de développer une approche et un prototype permettant la génération automatique de données de tests, a été atteint.

1.6 Recherches futures

Le présent travail constitue un premier pas à l'intérieur d'un vaste projet centré sur le génération de données de tests. Plusieurs limites ont été rencontrées durant son élaboration et constituent, inévitablement, des pistes de recherches futures en vue d'améliorer le présent système. Parmi ces améliorations, nous pouvons identifier particulièrement :

- l´élaboration d'un traitement plus efficace en présence d'indexation symbolique ;
- le traitement, dans un système orienté-objets, de caractéristiques telles que l´héritage multiple et les appels de constructeurs, qui ne sont pas traités dans le système actuel ;
- le traitement éventuel de chemins non-linéaires.

De plus, le système pourrait être étendu pour pouvoir tirer profit des avantages liés à des critères de couvertures plus évolués que la couverture des branches, comme par exemple la couverture des chaînes définitions-utilisations dans les programmes, approche reposant sur l'analyse de flux de données.

Finalement, une phase expérimentale plus imposante devrait être effectuée, ce qui permettrait de mieux valider notre approche, particulièrement sur des programmes de plus grande taille, ainsi que de mieux cerner les limites du présent système.

[9] BOYER, R.S., ELSPAS, B. et LEVITT, K.N. (1975). SELECT - a formal system for testing and debugging programs by symbolic execution. proc. of Int. Conf. on Reliable Soft., 234–245.

[10] CIMITILE, A., DE LUCIA, A. et MUNRO, M. (1995). Qualifying reusable functions using symbolic execution. IEEE proc. Second Working Conf. on Reverse Eng., 178–187.

[11] CIMITILE, A., DE LUCIA, A. et MUNRO, M. (1995). Identifying reusable functions using specification driven program slicing : a case study. IEEE proc. Int. Conf. on Soft. Maintenance, 124–133.

[12] CHUSHO, T. (1987). Test data selection and quality estimation based on the concept of essential branches for path testing. IEEE Trans. Soft. Eng., 509–517.

[13] CLARKE, L.A. (1976). A system to generate test data and symbolically execute programs. IEEE Trans. Soft. Eng., 215–222.

[14] COWARD, P.D. (1988). Symbolic execution systems - a review. Soft. Engineering Journal, 229–239.

[15] COWARD, P.D. (1991). Symbolic execution and testing. Information and soft. technology, 53–64.

[16] DEL FRATE, F., GARG, P., MATHUR, A.P. et PASQUINI, A. (1995). On the correlation between code coverage and soft. reliability. ISSRE'95 (Sixth int. symp. on soft. reliability engineering), 124–132.

[17] DEMILLO, R.A. (1991). Progress toward automated soft. testing. IEEE proc. of ICSE-13, 180–183.

[18] DEMILLO, R.A. et OFFUT, A.J. (1991). Constraint-based automatic test data generation. IEEE Trans. Soft. Eng., 900–910.

[19] FRANCKL, P.G. et WEYUKER, E.J. (1988). An applicable family of data flow testing criteria. IEEE Trans. Soft. Eng., 1483–1498.

[32] KOREL, B. et AL-YAMI, A.M. (1996). Assertion-oriented automated test data generation. IEEE proc. of ICSE-18, 71–80.

[33] KUNG, D., GAO, J., HSIA, P., TOYOSHIMA, Y., CHEN, C., KIM, Y.S. et SONG, Y.K. (1995). Developing an object-oriented software testing and maintenance environment. Comm. of the ACM, 75–87.

[34] LEJTER, M., MEYERS, S. et REISS, S.P. (1992). Support for maintaining object-oriented programs. IEEE Trans. Soft. Eng., 1045–1052.

[35] LENGAUER, T. et TARJAN, R.E. (1979). A fast algorithm for finding dominators in a flowgraph. ACM trans. on programming lang. and syst., 121–141.

[36] LINDQUIST, T.E. et JENKINS, J.R. (1988). Test-case generation with IOGen. IEEE Soft., 72–79.

[37] LUTZ, M. (1990). Testing tools. IEEE Soft., 53–57.

[38] MALAIYA, Y.K. (1995). Antirandom testing : getting the most out of black-box testing. ISSRE'95 (Sixth int. symp. on soft. reliability engineering), 86–95.

[39] VON MAYRHAUSER, A. (1992). Software testing : opportunity and nightmare. IEEE Int. Test Conf., 551–552.

[40] MILLER, E. (1985). Software testing technology : an overview. Handbook of Soft. Engineering, 359–379.

[41] MILLER, E. (1991). Software testing - the state of the practice. IEEE Int. Test Conf., p.1107.

[42] NEMHAUSER, G.L. et WOLSEY, L.A. (1988). Integer and combinatorial optimization. Wiley-Interscience series in discrete mathematics and optimization.

[43] NTAFOS, S. (1992). Software testing : theory and practice. IEEE Int. Test Conf., p.553.

[44] OFFUT, A.J. (1991). Unit testing versus integration testing. IEEE Int. Test Conf., 1108–1109.

Annexe A

"Partial Branch Coverability Conditions for Automatic Unit Test Data Generation"

Article écrit par E. Merlo, S. Lapierre, G. Savard, G. Antoniol, R. Fiutem et P. Tonella et soumis pour publication.

Partial Branch Coverability Conditions for Automatic Unit Test Data Generation

E. Merlo[1], S. Lapierre[1], G. Savard[1], G. Antoniol[2] , R. Fiutem[2], and P. Tonella[2]

[1] *GEGI, École Polytechnique*, C.P. 6079, Succ. Centre Ville. Montréal, Québec, Canada.
merlo@rgl.polymtl.ca, lapierre@casi.polymtl.ca, gilles@crt.umontreal.ca

[2] *IRST-Istituto per la Ricerca Scientifica e Technologica*, I-38050 Povo (Trento), Italy.
antoniol@irst.itc.it, fiutem@irst.itc.it, tonella@irst.itc.it

ABSTRACT

This paper presents an original approach to practical and feasible automatic unit test data generation based on the novel concept of simultaneous analysis of alternative paths defining a set of partially ordered path conditions. Conditions composing such alternative paths, defined in this paper as partial branch coverability conditions ($PBCC$), are produced while performing symbolic execution on execution trees. Solving the $PBCC$ using mixed-integer linear programming produces the input data to be used as test-cases.

Several strategies have been investigated to insure practicality. Methods used to attain such a goal include:

- reducing the number of test cases, which are needed to achieve the desired coverage, based on the concept of unconstrained arcs in a control flow graph;
- significantly reducing the difficulties originated by infeasible paths by producing and analyzing minimum path-length execution trees for such unconstrained arcs;
- reducing the difficulties originated by non-linear path conditions by considering alternative linear paths.

Execution trees are symbolically executed to produce partial branch coverability conditions ($PBCC$), which are then partially mapped into linear problems whose solutions correspond to the test data to be used as input to cover program branches. Partially mapping this problem into a linear optimization problem avoids infeasible and non-linear path problems, if a feasible linear alternate path exists in the same execution tree.

The presented approach has been implemented and tested on C-language programs. Preliminary results are encouraging and show that a high percentage of the program branches can be covered by the test data automatically produced.

KEYWORDS

Automatic test data generation, execution tree, mixed-integer linear programming, symbolic execution, partial branch coverability condition ($PBCC$), path infeasibility.

2.2 Symbolic-execution based generation of test data

Test data generation systems based on symbolic execution usually implies 3 distinct modules, which are:

- path generator
- symbolic executor together with a constraint generator
- inequality solver

In the path generation phase, the program's structure is statically analyzed and a given path selection strategy is applied in order to determine specific paths that will have to be processed to achieve coverage of a given testing criterion.

Symbolic execution is then performed on the resulting paths, one at a time. Symbolic execution could be described as a simulation of a real execution on a given path, with all input values left unknown and being replaced by symbolic values. All operations involving input values consequently are mapped into manipulations of the corresponding symbolic values, yielding symbolic expressions. During this symbolic execution, conditional predicates encountered along the execution path are accumulated into a Path Condition (PC) containing symbolic expressions of the variables composing each such conditional predicate. As a result, the PC contains a conjunction of all the distinct predicates encountered, expressed as a function of symbolic expressions representing inputs.

An inequality solver module is given the task of analyzing the resulting PC in order to deduce some symbolic values that will render it valid, thus finding some actual input values that will force a real execution of the program through the path of interest.

Since 1975, a number of approaches have been described to create a test case generator based on symbolic execution. To help understand our research interests and settings, here is a summary of such past symbolic execution system's main contributions and limitations:

A major contribution has undoubtedly been made in building the foundations of symbolic execution's theory and it's application to testing, with the EFFIGY system [25]. Past research has also contributed in identifying inherent problems with that technique and helped us better understand their implications. A lot of work still has to be made in that field, as suggested by the following problems and limitations encountered on previous systems.

The languages analyzed were mostly limited in regard of their complexity or were excessively simplified and generic, being only composed of basic operations. As an example, to our knowledge, no past or present systems which symbolically analyze pointers, whether to memory location (heap or stack) or to functions, has been described in the literature.

Function calls also seem to have been problematic and were only treated by a minority of past systems. Two main approaches have been employed and were identified as macro-expansion and lemma approach (see the SELECT system [12]). To our knowledge, no system accepted arbitrarily placed function calls, i.e. a call such as $if(2 * f(1, a, g(2)) \leq 0)$, which are frequently used in modern programs and which are dealt with in our system as explained in appendix B.

Given the above-mentioned limits of past test data generation systems, we have conceived an approach in view of improving automatic unit test data generation. Our approach presents the following characteristics:

- strategy to deal with infeasible paths
- treatment of arbitrarily placed function calls in expressions.
- treatment of pointers, function pointers, and dynamic memory operations.
- use of programmers interaction to solve symbolic indexing.
- treatment of programs written in a modern and widely used language (C).

3 TEST DATA GENERATION DESCRIPTION

In this section, we present an approach to achieve automatic unit test data generation. We describe the main algorithms used to implement the generation in section 4. An example will be used throughout this section in order to illustrate the presented concepts. The program *wc*, which is used as an example, is shown in figure 1 and has been taken from [18].

3.1 control flow graph

A control flow graph (CFG) for function f can be defined as follows :

$$Cfg = (V_{CFG}, E_{CFG})$$

where each node in V_{CFG} corresponds to a statement in the analyzed program and each edge in E_{CFG} corresponds to a flow of control between statements in V_{CFG}. The following holds:

$$(e =< v_1, v_2 > \in E_{CFG}) \rightarrow (v_1 \in V_{CFG}) \wedge (v_2 \in V_{CFG})$$

The program control flow graph for program *wc* is given in figure 2.

3.1.1 unconstrained edges Unconstrained edges were introduced in [1, 5, 10]. Using the definitions presented in [5], the set of unconstrained edges UE is defined as the set of edges in a CFG that are dominated by no other edge and that are post-dominated by no other edge, which corresponds to the following equation:

$$\begin{array}{c} < v_a, v_b > \in UE \Leftrightarrow \\ (\nexists < v_i, v_j > \in E_{CFG} \mid \\ (\forall p =<< v_{start}, v_n >, ..., < v_i, v_j >> \rightarrow < v_a, v_b > \in p)) \wedge \\ (\nexists < v_r, v_s > \in E_{CFG} \mid \\ (\forall q =<< v_r, v_s >, ..., < v_m, v_{stop} >> \rightarrow < v_a, v_b > \in q)) \end{array} \quad (1)$$

where $< v_a, v_b > \in E_{CFG}$ is an edge in the CFG , v_{start} and v_{stop} are the starting and ending edges in V_{CFG}.

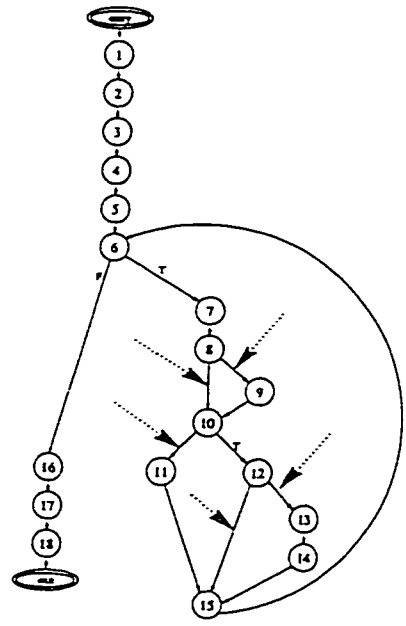


Figure 2: *CFG for wc program.*

The set of nodes is defined as follows:

$$\begin{array}{c} V_{ET} \stackrel{\text{def}}{=} \{< w_p, SubPathLeaves > \mid \\ (\exists p =< v_{start}, ..., v_i, ..., v_m >, v_i \in V_{CFG}) \wedge \\ (\forall i, (0 \leq i \leq m-1), < v_i, v_{i+1} > \in E_{CFG}) \wedge \\ (SubPathLeaves =< w_1, w_2, ..., w_n > \mid w_i \in E_{ET} \wedge \\ node - type(stm(w_i)) \in \{RETURN, STOP\} \end{array} \quad (2)$$

SubPathLeaves represents the leaves of the paths to be followed in the called functions from node $stm(w_p)$, and can be accessed through function *path-leaves* : $V_{ET} \rightarrow SubPathLeaves$.

Function $stm : V_{ET} \rightarrow V_{CFG}$ defines the correspondence between a node in the execution tree and its associated node in the control flow graph of the program as follows:

$$\begin{array}{c} stm = \{<< w_p, SubPathLeaves >, v_m > \mid \\ (\exists p =< v_{start}, ..., v_i, ..., v_m >, v_i \in V_{CFG}) \wedge \\ (\forall i, (0 \leq i \leq m-1), < v_i, v_{i+1} > \in E_{CFG})\} \end{array} \quad (3)$$

The set of edges in the execution tree is defined as:

$$E_{ET} \stackrel{\text{def}}{=} \{< w_1, w_2 > \mid (< stm(w_1), stm(w_2) > \in E_{CFG}) \wedge (w_1 \in V_{ET}) \wedge (w_2 \in V_{ET})\} \quad (4)$$

Intuitively, an execution tree is a tree representing multiple partial executions, which contain some common subpaths as a prefix and which differ at a given CFG branch.

Equation (2) defines V_{ET}, the set of nodes of ET, as a set of nodes w_p whose associated node in the CFG can be reached through an execution path from the starting node of the CFG (v_{start}) and ending at CFG node $stm(w_p)$.

Equation (4) states that E_{ET} is composed of edges whose starting and ending nodes have associated nodes in the CFG which form an edge of E_{CFG}

3.1.3 paths and partial-execution-tree leaves Let us define a path in an execution tree $ET = (V_{ET}, E_{ET})$ as follows:

$$path(w_m) =< w_1, ..., w_m > \mid ((w_i \in V_{ET}) \wedge (\forall i, (0 \leq i \leq m-1), < w_i, w_{i+1} > \in E_{ET})) \quad (5)$$

In this context, a path can be represented as a series of adjacent nodes in ET.

The function

$$max\_loops : V_{ET} \rightarrow N$$

can be defined as follows:

$$max\_loops(w) = \max_{u \in path(w)} \sum_{(z \in path(w)) \wedge (u \neq z)} \delta(stm(u), stm(z)) \quad (6)$$

where

$$\delta(x, y) = \begin{cases} 1 & \text{if x=y} \\ 0 & \text{otherwise} \end{cases}$$

The function $max\_loops(w)$ returns the maximum number of times that any instruction in the CFG has been encountered while traversing a particular path in the execution tree from the root to node w.

Using max_loops, function

$$et\_leaves : V_{CFG} \times N \rightarrow 2^{V_{ET}}$$

can be defined as

$$et\_leaves(v, k) = \{w \in V_{ET} \mid (stm(w) = v) \wedge (max\_loops(w) = k)\} \quad (7)$$

$ETL_{v,k} = et\_leaves(v,k)$ represents the set of leaves of the subtree of ET which contains all the partial executions ending in $v \in V_{CFG}$. Furthermore, each execution path contains at least one instruction which is encountered exactly k times and contains no instruction which is encountered more than k times.

As an example, $ETL_{12,1}$ for program *wc* with partial execution tree given in figure 3 would be the execution tree nodes in the following set: $\{20, 23, 35, 38\}$.

Let

$$et\_nodes : 2^{V_{ET}} \rightarrow 2^{V_{ET}}$$

where $2^{V_{ET}}$ is the power set of V_{ET}, be a function, which associates nodes belonging to a subset $V \subseteq V_{ET}$ to the nodes belonging to the subtree encountered while traversing ET from its root to nodes in V, and which is defined as follows:

$$et\_nodes(V) = \bigcup_{w \in V} \{z | z \in path(w)\}$$

$ETN_{v,k} = et\_nodes(et\_leaves(v,k))$ represents the nodes belonging to V_{ET} encountered while traversing ET from its root to all nodes in $ETL_{v,k}$.

In order to establish a relationship between the above defined concepts and our goal of branch coverage, let us define function $et\_leaves' : E_{CFG} \times N \rightarrow 2^{V_{ET}}$ as follows:

$$\begin{aligned} et\_leaves'(e,k) = \{w \in V_{ET} | \quad & (e =< v_1, v_2 >) \wedge \\ & (stm(w) = v_2) \wedge \\ & (\exists w' \in V_{ET} | < w', w > \in E_{ET}) \wedge \\ & (stm(w') = v_1) \wedge \\ & (max\_loops(w) = k)\} \end{aligned} \tag{8}$$

Using equation 8, the set $ETN'_{e,k}$, with $e =< v_1, v_2 >$, represents the nodes belonging to V_{ET} encountered while traversing ET from its root to all nodes in $et\_leaves'(e,k)$. $ETN'_{e,k}$ thus represents the nodes of V_{ET} encountered while traversing the execution tree from its root to CFG edge e, representing a program branch, and where each execution path contains at least one instruction which is encountered exactly k times along that path and where no instruction is encountered more than k times.

As an example, the partial execution tree for program *wc* computed to reach edge $< 12, 15 >$ considering $k = 2$ iteration corresponds the set of nodes $ETN'_{<12,15>,2}$ with label $\in \{1..39\}$ as shown in figure 3.

An interesting property of the function *et_leaves* is that it establishes a partial order over its range in the following way:

$$ETL_{v_1,k_1} \preceq ETL_{v_2,k_2} \Leftrightarrow (v_1 = v_2) \wedge (k_1 \leq k_2) \tag{9}$$

The corresponding partial execution tree is reported in figure 5. In this partial execution tree,

$$ETL_{5,1} = \{5\},$$

$$ETL_{5,2} = \{8\},$$

$$ETN_{5,1} = \{1, 2, 3, 5\},$$

$$ETN_{5,2} = \{1, 2, 3, 4, 6, 8\}.$$

Although $ETL_{5,1}$ precedes $ETL_{5,2}$, $ETN_{5,1}$ is not included in $ETN_{5,2}$, that is :

$$(ETL_{5,1} \preceq ETL_{5,2}) \not\Rightarrow (ETN_{5,1} \subseteq ETN_{5,2})$$

```
{
     int a, i ;
     ...
1 :  scanf("\%d", &a);
2 :  i = 0 ;
3 :  while(a > i)
4 :       i++ ;
5 :  printf("\%d", i) ;
     ...
}
```

Figure 4: *example program.*

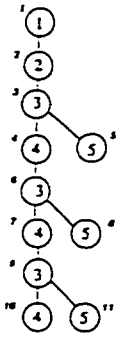


Figure 5: *partial execution tree for program of figure 4.*

In our opinion, the average length of the paths from the root to a node in $et\_leaves(v, k)$, for some v and some k, grows with k, while the size of the subtree of ET from the root to all nodes in $ETL_{v,k}$ for some v and some k, grows exponentially with k.

Since previous findings indicate that the difficulties originated by infeasible paths increase with the path length [44], it makes sense to use the partial order induced by $et\_leaves(v, k)$ to prioritize the

test data generation process, as described in section 3.2. These priorities privilege the resolution of paths which have a higher chance of being feasible.

An edge condition function

$$ec : E_{CFG} \rightarrow EC\_EXPR$$

is defined as a function taking an edge in the CFG and computing the condition under which control can follow such an edge.

Edge conditions are relevant and computed only in case of multiple edges from the same CFG node, as it happens for `if`, `switch`, `while`, `for`, and similar C-language constructs. For sequential flow of control, the single-edge condition is trivially $TRUE$ and is neither computed, nor stored in our system.

The computation of significant edge condition is performed, in our approach, using symbolic execution techniques as described in section 4. Extensions have been made to deal with C-language constructs such as function pointers, dynamic memory operations and arbitrarily placed function calls in expressions. Details on the symbolic execution and language coverage are presented in section 5.1.3.

3.2 Test data generation problem

The test data generation problem can be formulated as the satisfiability problem of finding values of the problem variables associated to program inputs which satisfy the following boolean expression:

$$PBCC_{CFG}(v,k) = \bigvee_{w \in et\_leaves(v,k)} pc(w) \tag{12}$$

where $PBCC_{CFG}(v,k)$ represents a logical disjunction between all the path conditions $pc(w)$ associated to nodes in $ETL_{v,k}$. Basically, equation 12 represents the logical condition associated to reaching node v in the CFG through all the paths which contain at least one node which is repeated exactly k times. This "node reachability condition", termed as a partial branch coverability condition ($PBCC$) in our work, is an original advancement with respect to previous approaches which considered one path at a time. In our approach, infeasible paths correspond to cases where $pc(w)$ is $False$ for some w. This is definitely not a problem while trying to solve $PBCC$ if some alternative path conditions $pc(w')$ for some w' can be satisfied by some inputs. Path conditions can be computed by symbolic execution, as described in section 4. For general references, see also section 2.2).

The boolean expression $PBCC_{CFG}(v,k)$ can be expressed as a function $PBCC_{ET}(w,v,k)$ on the execution tree, as follows:

$$PBCC_{CFG}(v,k) \leftrightarrow PBCC_{ET}(w_0,v,k) \tag{13}$$

```
(AND
      (AND
            (X2 ≠ EOF)
            (X1 ≠ '\t')
            (X1 ≠ ' ')
            (X1 ≠ '\n')
            (X1 ≠ EOF)
      )
      (OR
            (AND
                  (X2 ≠ '\n')
                  (X2 ≠ '\t')
                  (X2 ≠ ' ')
                  (X1 = '\n')
            )
            (AND
                  (X2 ≠ '\n')
                  (X2 ≠ '\t')
                  (X2 ≠ ' ')
            )
      )
)
```

Figure 6: *Partially simplified logical condition given by* $PBCC'_{ET}(1, < 12, 15 >, 1)$ *for wc example, with input data being represented by* X_1 *and* X_2. *Note that special characters '\n', '\t', ' ' and EOF are converted to their numerical ASCII equivalent before proceeding with the test data generation process.*

for example, of fewer 0-1 variables, shorter execution paths to be tested, and so on. The purpose of this paper has been limited to the study of the existence and the practical feasibility of a solution to the mixed-integer linear programming problem associated to unit test data generation.

A linear constraint c_i can also be represented by the triple $c_i = (T_i, r_i, b_i)$, where T_i is a set of terms defined as

$$T_i = \{< a_{i,j}, x_{i,j} > \mid a_{i,j} \in \Re, x_{i,j} \in VarSpace\},$$

$VarSpace$ is the set of possible linear problem variables, $r_i \in \{\leq, \geq\}$ is a relational operator, and $b_i \in \Re$ is a constant.

Projection functions over the three components of a constraints c_i can be defined as

$$\Pi_1(c_i) = T_i,\ \Pi_2(c_i) = r_i,\ \Pi_3(c_i) = b_i.$$

The linear problem consists of finding some values of the linear problem variables associated to program inputs which satisfy all linear constraints in $LC_{ET}(w_0, e, k)$ for a given branch e and some

m_c is a constant value large enough to make the constraint c relevant or irrelevant to the solution of the linear problem, depending on the value 0 or 1 of $var(u, v)$. The value of m_c is:

$$m_c = \begin{cases} \text{-MAX\_REAL} & \text{if } \pi_2(c) = \text{“} \leq \text{”} \\ \text{MAX\_REAL} & \text{if } \pi_2(c) = \text{“} \geq \text{”} \end{cases} \tag{19}$$

where MAX_REAL represents a proper large enough real number.

To partially map an edge condition into a set of linear constraints, function:

$$lp : EC\_EXPR \rightarrow LC$$

has been defined. Not all edge conditions can be converted into a set of linear constraints. If the conversion is not possible (and this depends only on the conditions written by programmers, since equations (17) and (18) are linear), function lp returns the empty set. Details about the partial conversion from arbitrary boolean expressions, which may include arbitrarily placed function calls, into a set of linear constraints can be found in appendix C.

As a final example, the linear problem associated to the logical expression given in figure 6 is given in figure 7, where M is a large constant (1000) and special characters $'\backslash n'$, $'\backslash t'$, $' \ '$ *and* EOF are converted to their numerical ASCII equivalent. In this figure, using the equations reported in appendix C, constraints $c1$ and $c2$ are obtained from condition $(X_2 \neq EOF)$ in figure 6, constraints $c3$ and $c4$ are obtained from condition $(X_1 \neq '\backslash t')$ in the same figure, and so on.

In the presented example, the following values have been computed as a result of the optimization process: X_1 = 'a', and X_2 = 'b'. Using these input values, program branch $< 12, 15 >$ can be effectively covered.

The advantage of having reduced the boolean expression problem to a linear one is that we can take advantage of linear optimization techniques [34] to solve this problem in an efficient way. On the other hand, not all boolean satisfiability problems can in general be converted into a linear one. Approaches to generate test data in the presence of non-linear expressions can be found in [7, 26, 38].

A purpose of this paper is to study the program branch coverage practically attainable using such a problem reduction approximation.

Equation (17) takes into account the possible existence of alternative paths reaching edge e in the CFG. To generate test data, it is sufficient to satisfy one of the alternative paths only.

Equation (18) specifies that alternative execution paths are mutually exclusive for test data generation, in agreement with the semantics of execution of a sequential program.

Both equations (17) and (18) can be implemented in an efficient way by traversing the execution tree ET and by generating new variables $var(w, u)$ and new constraints only when the number of successors of a given node w is greater or equal two (i.e., when an `if`, `switch`, `while`, `for`, or a similar C-language construct is encountered in the CFG).

Given $e =< v_1, v_2 >$, an edge of the CFG to be covered, the solution process fails only if:

1. All paths between $w_0 \in V_{ET}$ and nodes in $et\_leaves'(v_2, k)$ are non-linear, or
2. all linear paths between $w_0 \in V_{ET}$ and nodes in $et\_leaves'(v_2, k)$ are infeasible

Results presented in section 5.2 show that these failure conditions rarely happen in the programs we analyzed.

4 IMPLEMENTATION DETAILS

Given the problem formulated in the preceding section, here are presented the main algorithms and the descriptions of data structures required to achieve automatic test data generation.

4.1 Identifier Representation

Identifiers in a program can be local or global variables, parameter variables, functions or structure attributes. In our system, identifiers are represented by pairs, where each unique identifier can be unambiguously specified, with all possible pairs forming the set of possible identifiers $IdentifierSpace$, with:

$$IdentifierSpace = \{< func\_id, var\_id >\}$$

where $func\_id \in N^+$ and $var\_id \in N^+$ are integer values and:

$$FUNC\_ID : IdentifierSpace \rightarrow func\_id, \ with$$

$$FUNC\_ID(id) = \begin{cases} >= 1 & \text{if } id \text{ is a function,} \\ & func\_id \text{ then is the unique integer associated to that function} \\ 0 & \text{otherwise (if not a function)} \end{cases}$$

$$VAR\_ID : IdentifierSpace \rightarrow var\_id, \ with$$

$$VAR\_ID(id) = \begin{cases} >= 1 & \text{if } id \text{ is a variable or structure attribute,} \\ & var\_id \text{ then is the unique integer associated to that variable/attribute} \\ 0 & \text{otherwise} \end{cases}$$

Constraints on identifier representations are given in table 1.

Identifier category	*func_id*	*var_id*
user function/main	> 0	0
struct. attribute	0	> 0
global variable	0	> 0
local variable/parameter	> 0	> 0

Table 1: *Examples of identifier representation.*

4.2 Data Values Representation

Current values of variables through symbolic execution are represented as a $State$, with $State = \{VarVal\}$, a $State$ being a collection of $VarVal$ which are associations between variables and their current values or fields (structure attributes or array elements). A $VarVal$ is a 3-tuple $< var, val, fields >$ with

$$var \in IdentifierSpace$$

$$val \in Expression(current\ value\ of\ var)$$

$$fields \in State$$

The following functions will be needed by the algorithms:

$$VAR : State \rightarrow IdentifierSpace, \ with \ VAR(< a, b, c >) = a$$

$$VAL : State \rightarrow Expression, \ with \ VAL(< a, b, c >) = b$$

$$FIELDS : State \rightarrow State, \ with \ FIELDS(< a, b, c >) = c$$

4.3 Algorithms

The algorithms for symbolic executions and for the construction of the linear problem associated to an iteration limit k are presented as appendices to this paper. They are presented for the sake of detailed understanding of the presented approach, but the reader could skip them without compromising the understanding of the next sections.

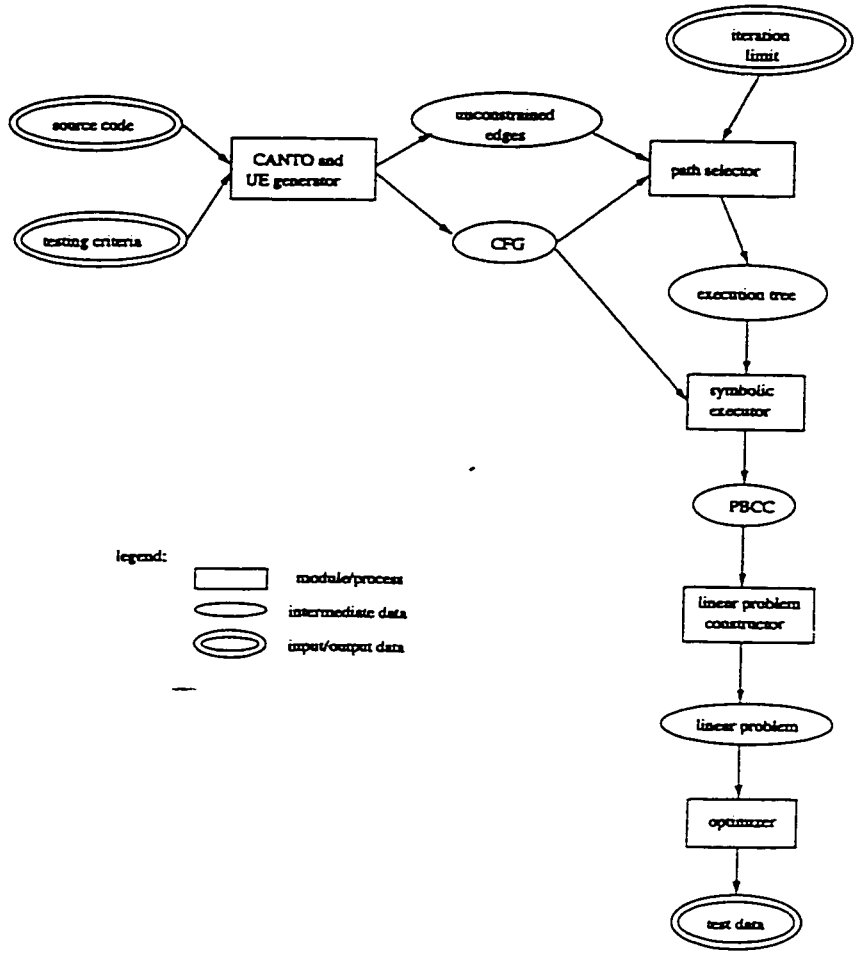


Figure 8: *TAO system architecture*

The path selector module is then given the task of computing execution trees ET in order to attain each CFG edge in the unconstrained edges set UE, considering a given number of iterations. The production of ET for a CFG edge e for a number of iterations k is thus equivalent to computing the tree with a set of nodes equivalent to $ETN'_{e,k}$. This computation is achieved with a simple breadth-first search through a unit CFG, with goal edge e, using node traversal counters to limit the number of iterations to k.

In our system, programmers may interactively impose the number of iterations to be analyzed during path selection in case of fixed-length loops (such as in initializations), which basically corresponds to a fixed-length loop unfolding operation. This operation does not affect the performance of the algorithm, facilitates test-case generation (which would otherwise be confronted with more infeasible paths) and is very easily implemented, especially for initialization loops.

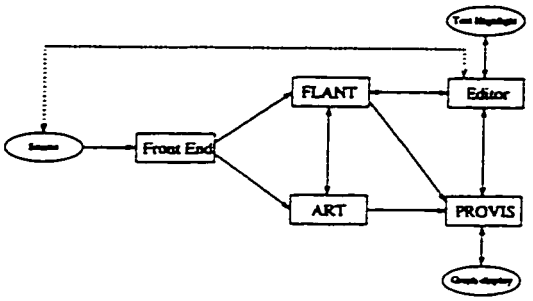


Figure 9: Architecture of CANTO.

5.1.3 Symbolic Executor. *TAO*'s symbolic executor traverses the unit CFG along the paths specified in the execution tree ET given by the path selector. Symbolic execution proceeds along each path given by the execution tree, constantly updating the modifications made to the tested unit's variables and decorating the corresponding execution tree structure with the appropriate constraints encountered at branching points along the paths. For further details about the principles behind symbolic execution, see references given in section 2.2. In our system, functionality has been added to enable the processing of complex constructs such as symbolic indexing, pointers and pointer arithmetic, dynamic allocation and deallocation, function pointers, bitwise operations and recursive functions.

At this stage of our research, a simple strategy has been selected to treat symbolic indexing. Our system detects symbolic indexing, informs and relies on the user in order to determine an exact array index to be used and proceeds with normal symbolic execution, using this index. When an exact index is used that renders incoherent the corresponding execution, a new execution is generated and another value of index is asked to the human tester.

In symbolic execution, pointers and function pointers are computed along the unique path followed through the unit CFG and, consequently, all references associated to pointers are symbolically known along such a path, even though they are approximate since infeasibility will only be detected later.

Our symbolic executor's complexity is a linear function of the number of nodes in the execution tree. The result of symbolic execution is a partial branch coverability condition ($PBCC$) which represents a boolean condition corresponding to $PBCC_{ET}(w_0, v, k)$ and computed by equation (14) during symbolic execution along the paths of the execution tree, given as a function of input values.

5.1.4 Linear Problem Construction and Optimizer. In our system, the inequality solver module relies on linear programming. In order to try to find a solution to the accumulated boolean conditions ($PBCC$), the partial branch coverability conditions are transformed into a linear set of constraints and formulated as input files to an optimization tool. This tool tries to solve the linear problem given and informs our system of the solutions found or of it's inability to optimize the given problem.

5.2 Experimental setup and results

Here are presented preliminary results that will be discussed in the following section.

A total of 12 C-language programs containing 41 units totalizing 1464 LOC have been analyzed. Some of these programs were selected because they were mentioned in the literature (*wc* [18], *patternmatcher* [17] and *trityp* [16]) and others because they contained precise characteristics that are difficult to analyze and that we were interested in investigating, such as :

- symbolic indexing
- pointers and pointer arithmetic
- function pointers

One program (*taotest*) belongs to the test set for our system. General characteristics of these programs are shown in table 2, namely their size in lines of code, their complexity according to McCabe's cyclomatic complexity metric [4], and the presence of precise characteristics, such as the presence of pointers, function pointers or symbolic indexing within some units of each program.

Program	LOC	McCabe metric	ptrs.	func. ptrs.	symb. indexing
wc	39	6	-	-	-
patternmatcher	77	8	X	-	-
prog6	68	6	-	-	-
taotest	239	74	X	X	-
minpath	194	64	X	-	X
quicksort	95	20	X	-	X
histogramme	45	6	-	-	X
what	130	3	X	-	-
trityp	62	16	-	-	-
exptree	389	190	X	-	X
nombre	67	7	-	-	-
inves	57	3	-	-	-
	total : 1462	average : 33.6	-	-	-

Table 2: *Programs analyzed by TAO.*

The programs were first parsed and converted into an intermediate control flow graph language by CANTO [3]. Programs were then partitioned into functional units in order to proceed with unit testing. The unconstrained edges were afterwards generated and used as goals for our path selector module. As a first result, the CFGs of the units analyzed contained a total of 722 edges. Computation of the unconstrained edges identified 131 edges (as reported in table 3) that had to be covered in order to ensure complete unit branch coverage. This shows a very significant reduction of 82% in the number of edges that had to be considered by test data generation while still potentially attaining full unit branch coverage.

The path selector module computed the execution trees that were to be analyzed through the symbolic executor. Given UE, the overall set of unconstrained edges in all unit CFGs, and k, a number of iteration, we can define

$$N_{ET}(k) = \frac{\sum_{e \in UE} card(ETN_{e,k})}{card(UE)}$$

$$N_c(v) = \frac{\sum_{e \in UE} card(LP_{CFG}(e,k))}{card(UE)}$$

and is the average number of linear constraints generated in the linear problems for all the unconstrained edges and for all units of that program and for a given number of iterations.

Problems with fixed number of iterations k were analyzed separately, starting from problems obtained with k=0 and extending to k=1 and 2. Table 3 shows the percentage ($\%_c$) of covered unconstrained edges for each program, considering all it's units, and for each number of iterations k analyzed, with k=0, 1 and 2. Note that $\%_c$, for a given k, is given as an added coverage over the coverage already achieved with lower values of k.

As final results, table 4 shows the unconstrained edges coverage achieved for all analyzed programs and the total time needed to achieve such coverage. To clarify the result presentation, UE coverage and time results were added for all units of each program. Time performances given represent CPU time needed to complete the process described above, from abstract syntax tree representation to final test data generation. Experiments were conducted on a PentiumPro 166Mhz processor with 64Mb RAM and running Linux operating system.

Program	UE coverage achieved for all units (%)	Time required (sec.)
wc	100	1.15
patternmatcher	100	1.48
prog6	100	0.68
taotest	100	11.23
minpath	94	14.97
quicksort	100	2.70
histogramme	100	0.78
what	100	1.29
trityp	94.7	12.67
exptree	100	36.5
nombre	100	0.63
inves	100	0.07

Table 4: *Total UE coverage and associated time performances for unit test data generation for the 12 analyzed programs.*

An interesting result is also given in figure 10 : it shows the total test-case generation time (in seconds) as a function of the coverage attained (in percentage). The 12 programs analyzed are presented on the same figure and are clustered as three separate groups for which we have have given the range of corresponding McCabe's cyclomatic complexity metric.

6 DISCUSSION

In [44], it is reported that a path's infeasibility tends to increase exponentially as a function of it's length. In order to decrease path infeasibility problems, our search strategy makes use of the partial order on the number of iterations and of the implications of unconstrained edges in such a way that the search conducted in the unit CFG is stopped when the goal unconstrained edge is reached, thus limiting the lengths of the selected paths. To our knowledge, no other test data generation system made use of the implications of a path's length to conduct their path selection strategy. In

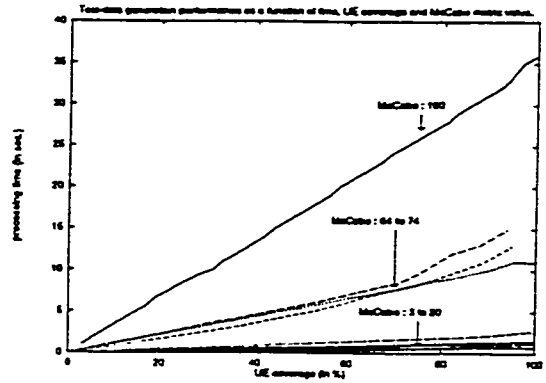


Figure 10: *Time performances as a function of coverage achieved for the 12 programs, with McCabe's cyclomatic complexity metric value given for program groups.*

fact, path selection has mostly been aimed at reducing the number of analyzed paths (as in [42]), an approach in which path infeasibility is not considered. The approach we implemented, which is based on [5], has the double advantage of significantly minimizing the number of paths to be considered (reduction of 82 % in our experiments) and of limiting path lengths, therefore reducing difficulties associated with path infeasibility.

The use of execution trees is another major factor in decreasing problems related to path infeasibility. In [43], it has been proven that determining the feasibility of a single program path is an undecidable problem. The use of execution trees helps us to get around this problem by potentially providing alternative feasible paths which will lead to a solution, thus overriding undecidability problems linked with any single path within the tree. This is achieved by considering sets of paths together and by considering a partial branch coverability condition.

Execution trees play a similar role in helping us avoiding problems related to non-linear set of constraints. Using an equation solver module based on linear programming, our system would not be able to solve any single non-linear set of constraints. But as mentioned earlier, use of execution trees provides us with alternatives that potentially help us avoid non-linear problems. Furthermore, it is our opinion that non-linear problems can be mostly avoided by using execution trees; inability to solve an execution tree due solely to non-linear problems can only be caused if each path within that tree involves a non-linear problem. In our opinion, the probability for an execution tree to contain exclusively non-linear paths is very low, as supported by research data on frequency of non-linear problems in computer programs [11, 13].

The preliminary experiments we conducted show a very high degree of unconstrained edges coverage. In fact, only 1.5% of all the unconstrained edges were not covered by our analysis, which represents 2 unconstrained edges out of 131. In one case (*trityp*), complete coverage was not possible due to a clearly infeasible path produced by contradicting sequencing of assignments and tests on internal variables. In the other case (*minpath*), the not-covered unconstrained edge was part of a loop that had to be performed 5 times in order to become valid. Our analysis was limited to the treatment of 0,1 and 2 iterations and we therefore could not automatically solve the above mentioned problem.

unit coverages in the 94%-100% range are very good. These results are very encouraging. Being based on unit testing, it is our opinion that our system would easily scale to process bigger systems composed of a larger number of units.

7 FUTURE RESEARCH

After having presented our test data generation approach and having discussed interesting experimental results, we present a number of interesting ideas and goals for future research.

Two major future research concerns are to accumulate data on the effectiveness of test data generation for larger systems and to accumulate data on frequency and complexity of symbolic indexing, in order to better understand the problem and to be in better position for finding a practical solution.

Other interesting points of future work include:

- increase user friendliness of the system (graph or tree displays)
- integrate data-flow testing criterion to conduct test data generation [39, 21]
- explore possibilities for processing non-linear equations (as mentioned in section 3.2)
- establish links with dynamic approaches when symbolic execution loses efficiency [26]
- extend the approach to deal with object-oriented languages such as C++
- increase the effectiveness of the test data generated by relying on approaches such as assertion testing [27], mutation testing [15, 16] or neuronal networks [2].

8 CONCLUSIONS

We have presented an approach for automatic unit test data generation which is based on the concept of partial branch coverability conditions and which makes use of mixed-integer linear programming, execution trees, and symbolic execution. Our approach improves practicality of test data generation by reducing the number of test data needed to achieve branch coverage and by substantially reducing the effects of infeasible paths and of non-linear set of constraints.

Experiments have been conducted on C-language programs and show a very high degree of unit branch coverage with test data automatically generated by our system.

ACKNOWLEDGEMENTS

We thank Charles Leduc and Christian Meunier, from École Polytechnique of Montréal, who helped in building *TAO*'s expression simplifier and unconstrained edges analyzer.

REFERENCES

[1] H. Agrawal. "Dominators, Super Blocks, and Program Coverage", *POPL'94:Symp. on Principles of Programming Languages*, 1994, pp.25–34.

[2] C. Anderson, A. von Mayrhauser and T. Chen. "Assessing neural networks as guides for testing activities", *IEEE proc. of METRICS'96*, 1996, pp.155–165.

[3] G. Antoniol, R. Fiutem, G. Lutteri, P.Tonella S. Zanfei and E. Merlo. "Program Understanding and Maintenance with the CANTO Environment", To appear In *Proceedings of the International Conference on Software Maintenance*, Sept 28-30 Oct 1-3, Bari, 1996.

[4] B. Beizer. "Software testing techniques, second edition", International Thomson computer press, 1990.

[5] A. Bertolino and M. Marré. "Automatic generation of path covers based on the control flow analysis of computer programs", *IEEE Trans. Soft. Eng.*, dec. 1994, pp.885–898.

[6] J. Bicevskis, J. Borzovs, U. Straujums, A. Zarins and E. F. Miller. "SMOTL - A system to construct samples for data processing program debugging", *IEEE Trans. Soft. Eng.*, jan. 1979, pp.60–66.

[7] R. S. Boyer, B. Elspas and K. N. Levitt. "SELECT - a formal system for testing and debugging programs by symbolic execution", *proc. of Int. Conf. on Reliable Soft.*, april 1975, pp.234–245.

[8] A. Cimitile, A. De Lucia and M. Munro. "Qualifying reusable functions using symbolic execution", *IEEE proc. Second Working Conf. on Reverse Eng.*, july 1995, pp.178–187.

[9] A. Cimitile, A. De Lucia and M. Munro. "Identifying reusable functions using specification driven program slicing: a case study", *IEEE proc. Int. Conf. on Soft. Maintenance*", oct. 1995, pp.124–133.

[10] T. Chusho. "Test data selection and quality estimation based on the concept of essential branches for path testing", *IEEE Trans. Soft. Eng.*, may 1987, pp.509–517.

[11] L. A. Clarke. "A system to generate test data and symbolically execute programs", *IEEE Trans. Soft. Eng.*, sep. 1976, pp.215–222.

[12] P. D. Coward. "Symbolic execution systems - a review", *Soft. Engineering Journal*, nov. 1988. pp.229–239.

[13] P. D. Coward. "Symbolic execution and testing", *Information and soft. technology*, 1991. pp.53–64.

[14] F. Del Frate, P. Garg, A. P. Mathur and A. Pasquini. "On the correlation between code coverage and soft. reliability", *ISSRE'95 (Sixth int. symp. on soft. reliability engineering)*, 1995, pp.124–132.

[15] R. A. DeMillo. "Progress toward automated soft. testing", *IEEE proc. of ICSE-13*, 1991. pp.180–183.

[16] R. A. DeMillo and A. J. Offutt. "Constraint-based automatic test data generation", *IEEE Trans. Soft. Eng.*, sep. 1991, pp.900–910.

[17] P. G. Franckl and E. J. Weyuker. "An applicable family of data flow testing criteria", *IEEE Trans. Soft. Eng.*, oct. 1988, pp.1483–1498.

[18] K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance", *IEEE Trans. Soft. Eng.*, 17(8), 1991, pp.751–761.

[19] T. W. Gary. "Software testing, the state of the practice", *IEEE Int. Test Conf.*, 1991, p.1106.

[20] R. L. Glass. "Persistent software errors", *IEEE Trans. Soft. Eng.*, march 1981, pp.162–168.

[21] R. Gupta and M. L. Soffa. "Priority based data flow testing", *IEEE ICSM*, 1995, pp.348–357.

[22] W. E. Howden. "An evaluation of the effectiveness of symbolic testing", *Soft.-Practice and Experience*, 1978, pp.381–397.

[23] D. C. Ince. "The automatic generation of test data", *The Computer Journal*, 1987, pp.63–69.

[24] R. Jasper, M. Brennan, K. Williamson and B. Currier. "Test data generation and feasible path analysis", *Int. symp. on soft. testing and analysis*, 1994, pp.95–107.

[25] J. C. King. "Symbolic execution and program testing", *ACM Programming Languages*, july 1976, pp.385–394.

[26] B. Korel. "Automated software test data generation", *IEEE Trans. Soft. Eng.*, aug. 1990, pp.870–879.

[27] B. Korel and A. M. Al-Yami. "Assertion-oriented automated test data generation", *IEEE proc. of ICSE-18*, 1996, pp.71–80.

[28] T. E. Lindquist and J. R. Jenkins. "Test-case generation with IOGen", *IEEE Soft.*, jan. 1988, pp.72–79.

[29] M. Lutz. "Testing tools", *IEEE Soft.*, may 1990, pp.53–57.

[30] Y. K. Malaiya. "Antirandom testing : getting the most out of black-box testing", *ISSRE'95 (Sixth int. symp. on soft. reliability engineering)*, 1995, pp.86–95.

[31] A. von Mayrhauser. "Software testing : opportunity and nightmare", *IEEE Int. Test Conf.*, 1992, pp.551–552.

[32] E. Miller. "Software testing technology : an overview", *Handbook of Soft. Engineering*, 1985, pp.359–379.

[33] E. Miller. "Software testing - the state of the practice", *IEEE Int. Test Conf.*, 1991, p.1107.

[34] G. L. Nemhauser and L. A. Wolsey. "Integer and combinatorial optimization", Wiley-Interscience series in discrete mathematics and optimization, 1988.

[35] S. Ntafos. "Software testing : theory and practice", *IEEE Int. Test Conf.*, 1992, p.553.

[36] A. J. Offutt. "Unit testing versus integration testing", *IEEE Int. Test Conf.*, 1991, pp.1108–1109.

[37] A. J. Offut and S. D. Lee. "An empirical evaluation of weak mutation", *IEEE Trans. Soft. Eng.*, may 1994, pp.337–344.

A CONTROL FLOW GRAPH MANIPULATION FUNCTIONS

In this appendix, we describe the functions that need to be defined on the nodes of the CFG in order for the symbolic execution algorithms to be complete.

As described earlier, a control flow graph (CFG) can be defined as follows :

$$Cfg = (V_{CFG}, E_{CFG})$$

The nodes in V_{CFG} are typed, with function NODE_TYPE: V_{CFG} →CFG_TYPES returning their type information, and

CFG_TYPES = $Statement \cup Expression$,

where $Statement =$
$\{Assign, If, While, Return, Start, Stop, PreInc, PostInc, PreDec, PostDec, UserFctCall,$
$SystemFctCall\}$ and $Expression = \{Unary\_exp, Binary\_exp, Const, Symbol, VarVal, Identifier\}$.

The following functions are defined in order to enable us to manipulate the Cfg nodes:

Functions GET_START_NODE and PARAM are directly applied on a Cfg structure in order to get the starting node of a Cfg and to gain access to declared parameters on that Cfg.

GET_START_NODE:$Cfg \rightarrow Start$, returns the starting node of function Cfg.

PARAM:$Cfg \times N \rightarrow IdentifierSpace$, where PARAM$(cfg, n)$ returns the n^{th} parameter expression declared in the header of function cfg, which corresponds to an identifier expression (see section 4.1).

Functions LHS and RHS enable us to manipulate the newly defined variable space (LHS) in an assignment or a post-pre/inc-decrementation and the new computed value to be assigned (RHS) in an assignment.

LHS: $V_{CFG} \rightarrow Expression$, defined as follows:

$$\text{LHS}(v) \stackrel{\text{def}}{=} \begin{cases} \text{the left hand-side of } v & \text{if NODE\_TYPE}(v) \in \\ & \{Assign, Preinc, PostInc, PreDec, PostDec\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

RHS:$V_{CFG} \rightarrow Expression$, defined as follows:

$$\text{RHS}(v) \stackrel{\text{def}}{=} \begin{cases} \text{the right hand-side of } v & \text{if NODE\_TYPE}(v) = Assign \\ \text{undefined} & \text{otherwise} \end{cases}$$

Functions LEFT_CHILD and RIGHT_CHILD return the left and right component in a binary expression, while CHILD returns the unique child of an unary expression. OP is defined as returning the identity of the operator involved in an unary or binary expression.

LEFT_CHILD: $V_{CFG} \rightarrow Expression$, defined as follows:

$$\text{LEFT\_CHILD}(v) \stackrel{\text{def}}{=} \begin{cases} \text{the left child of } v & \text{if NODE\_TYPE}(v) = \textit{Binary\_exp} \\ \text{undefined} & \text{otherwise} \end{cases}$$

RIGHT_CHILD: $V_{CFG} \rightarrow \textit{Expression}$, defined as follows:

$$\text{RIGHT\_CHILD} \stackrel{\text{def}}{=} \begin{cases} \text{the right child of } v & \text{if NODE\_TYPE}(v) = \textit{Binary\_exp} \\ \text{undefined} & \text{otherwise} \end{cases}$$

OP: $V_{CFG} \rightarrow \textit{Operator}$,
with $\textit{Operator} = \{+, -, *, /, \%, \rightarrow, ., [], AND, OR, EQ, NEQ, LEQ, LT, GEQ, GT\}$ and defined as follows:

$$\text{OP}(v) \stackrel{\text{def}}{=} \begin{cases} \text{the operator of } v & \text{if NODE\_TYPE}(v) \in \{\textit{Binary\_exp}, \textit{Unary\_exp}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

CHILD: $V_{CFG} \rightarrow \textit{Expression}$, defined as follows:

$$\text{CHILD}(v) \stackrel{\text{def}}{=} \begin{cases} \text{the unique child of } v & \text{if NODE\_TYPE}(v) = \textit{Unary\_exp} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Functions TEST and FALSE are defined on control transfer statements, namely If and $While$ statements. TEST returns the predicate expression to be tested during execution while FALSE returns the identity of the statement taken during execution if the control predicate evaluates to false.

TEST: $V_{CFG} \rightarrow \textit{Expression}$, defined as follows:

$$\text{TEST}(v) \stackrel{\text{def}}{=} \begin{cases} \text{the test expression of } v & \text{if NODE\_TYPE}(v) \in \{If, While\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

F_CHILD: $V_{CFG} \rightarrow \textit{Statement}$, defined as follows:

$$\text{F\_CHILD}(v) \stackrel{\text{def}}{=} \begin{cases} \text{the statement following } v \text{ if the test of } v \text{ evaluates to FALSE} & \text{if NODE\_TYPE}(v) \in \{If, While\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

In order to gain access to the identity of the called function, function ACCESS_PATH has been defined. As an example, given a call statement resembling $*(fptr[2])(a, b, c, d)$ involving an element within an array of function pointers, ACCESS_PATH would return $*(fptr[2])$.

ACCESS_PATH: $V_{CFG} \rightarrow \textit{Expression}$, defined as follows:

$$\text{ACCESS\_PATH}(v) \stackrel{\text{def}}{=} \begin{cases} \text{the expression to be evaluated to determine the fct/method to be called} & \text{if NODE\_TYPE}(v) = \textit{UserFctCall} \\ \text{undefined} & \text{otherwise} \end{cases}$$

ARG gives access to each argument expression used in a call statement. In the above mentioned example,

ARG$(*(fptr[2])(a,b,c,d),2)$ would return information b as being the second argument of function call $*(fptr[2])(a,b,c,d)$.

ARG: $V_{CFG} \times N \rightarrow$ *Expression*, defined as follows:

$$\text{ARG}(v,n) \stackrel{\text{def}}{=} \begin{cases} \text{the } n^{th} \text{ argument expression to be passed in a function call } v & \text{if NODE\_TYPE}(v) = UserFctCall \\ \text{undefined} & \text{otherwise} \end{cases}$$

RET_VAL is defined of *Return* statements and returns the expression to be computed and returned while executing the statement.

RET_VAL: $V_{CFG} \rightarrow$ *Expression*, defined as follows:

$$\text{RET\_VAL}(v,n) \stackrel{\text{def}}{=} \begin{cases} \text{the expression to be computed and returned in } v & \text{if NODE\_TYPE}(v) = Return \\ \text{undefined} & \text{otherwise} \end{cases}$$

- *base* : when dealing with structures, *base* gives a reference to the $VarVal$ corresponding to the base variable within which a field is to be specified, for example the $VarVal$ associated to a when accessing *a.price*.

- vs_g and vs_l : the sets of global-level and local-level variables currently computed and accessible (for local-level).

- *path_leaves* : vector of nodes which correspond to the execution tree leaves of the functions which will be called in order to execute node n. *path_leaves* results from the computation done in COMPUTE_EXECUTION_TREE_NODES(v, k). *path_leaves* can be empty if n doesn't perform a function call, or can be composed of numerous execution tree leaves if n performs numerous calls, for example if n is an assignation statement resembling $x = f(2 + g(h(1) * 3))$, which directly implies 3 function calls in the order h, g then f; in this example, a path will be identified within h, g and f and the leaf of that path in the corresponding execution tree will be associated to *path_leaves* for node v.

- *computed_val* : is a parameter passed by address and which will be used to return the computed value for the current node n. For example, if n is the binary expression 2+3, parameter *computed_val* will be used to return the computed value 5.

The function EVAL_NODE is also defined to return an expression. This expression corresponds to the logical condition computed in order to execute node v. It has to be noted that this expression takes into account all the conditions encountered while evaluating node v. For example, if v is the assignation statement $x = f(2 + g(h(1) * 3))$, the conditional expression computed for node v will be a conjunction of the conditions accumulated while symbolically executing the paths ending at the nodes of *path_leaves* within the execution trees of functions h, g and f.

Function EVAL_NODE makes use of other functions in order to perform symbolic execution. For completeness, these function's algorithms are given immediately following the algorithm of EVAL_NODE.

Note that in the following algorithm, in order to express that an argument is of no consequence when performing a function call, we will be using a generic argument n_a which stands for "not applicable". Function EVAL_NODE can be defined as follows (for further explanations, consult the comments given directly into the algorithms):

Function EVAL_NODE

(in $n, o{:}V_{CFG}$; $base : VarVal$; $vs_g, vs_l : State$; $VAR\ path\_leaves : 2^{V_{ST}}$; VAR $computed\_val : Expression$): $Expression$;
begin
  {*in case of an assignation statement, e.g. "lhs = rhs ;"*}
  if(NODE_TYPE(n) = *Assign*) then begin
    {*modify the value of lhs to be the computation of rhs*}
    {*consider the conditions potentially encountered while computing lhs and rhs*}
    $cond\_lhs := TRUE$
    $cond\_rhs := TRUE$
    {*evaluate the symbolical memory location of the lhs*}
    lhs := GET_ADDRESS(LHS(n),$n_a, vs_g, vs_l, path\_leaves, cond\_lhs$)
    {*evaluate the current value of the rhs*}
    rhs := GET_VALUE(RHS(n),$n_a, vs_g, vs_l, path\_leaves, cond\_lhs$)

```
    SET_VAL(lhs, rhs)
    {the returned value of this statement is rhs. Used in cases like "a = (b = c);"}
    computed_val := rhs
    return(cond_lhs ∧ cond_rhs)
 end

 {in case of an pre-post/inc-decrementation statement, }
 {e.g. "--lhs;" or "++lhs" or "lhs--" or "lhs++"}
 if(NODE_TYPE(n) ∈ {PreInc, PostInc, PreDec, PostDec}) then begin
    {inc/decrement the value at address lhs and return the value before/after that modification}
    {consider the conditions potentially encountered while computing lhs}
    {note : pointer inc/decrementation is handled by the INC function}
    cond_lhs = TRUE
    lhs := GET_ADDRESS(LHS(n),n_a, vs_g, vs_l, path_leaves, cond_lhs)
    oldval := GET_VALUE(LHS(n),n_a, vs_g, vs_l, path_leaves, n_a)
    if(NODE_TYPE(n) ∈ {PreInc, PostInc}) then
       newval := INC(oldval, 1)
    else
       newval := INC(oldval, -1)
    if(NODE_TYPE(n) ∈ {PreInc, PreDec}) then begin
       SET_VAL(lhs, newval)
       computed_val := newval
    end else begin
       computed_val := oldval
       SET_VAL(lhs, newval)
    end
    {the returned value of this statement is newval. }
    {Used in cases like "a = b++;" or "if(--b > c)"}
    return(cond_lhs)
 end

 {in case of a control transfer statement, e.g. "if()..." or "while()..."}
 if(NODE_TYPE(n) ∈ {If, While}) then begin
    {consider the conditions potentially encountered while computing test}
    cond_test = TRUE
    test := GET_VALUE(TEST(n),n_a, vs_g, vs_l, path_leaves, cond_test)
    {evaluate the current value of the test expression and return that value , possibly negated }
    {depending on the path to be followed, as the condition for edge "n → o" to be taken}
    if(F_CHILD(n)=o)
       return((test=FALSE) ∧ cond_test)
    else
       return((test=TRUE)) ∧ cond_test)
 end

 {in case of a user function call statement}
 if(NODE_TYPE(n) = UserFctCall) then begin
    {cond_call will accumulate all the conditions obtained while setting the call}
    cond_call := TRUE
    {map the argument's values to the parameters}
```

$vs'_l := \{\}$
∀ arguments *arg* begin
{Get a reference to the i^{th} parameter symbolic variable (while creating it)}
newparam := GET_ADDRESS(PARAM(cfg, *arg*),n_a, vs_g, vs'_l, *path_leaves*, n_a)
cond := TRUE
{Get the current value of the i^{th} argument of the function call}
newarg := GET_VALUE(ARG(n, *arg*),n_a, vs_g, vs_l, *path_leaves*, *cond*)
cond_call := *cond_call* ∧ *cond*
{Set the initial value of the parameter to be the same as the argument's}
SET_VAL(*newparam*, *newarg*)
end
compute the path to be followed in the execution tree of the called function
$V = \{v \mid v \in path(FIRST(path\_leaves))\}$
$E = \{e =< v_1, v_2 > \mid v_1, v_2 \in V \wedge < stm(v_1), stm(v_2) > \in E_{CFG}\}$
$path\_leaves = path\_leaves - FIRST(path\_leaves)$
{execute the called function/method along that path}
$ec := P_{ET}(V, E, ROOT(V, E), vs_g, vs'_l, computed\_val)$
{the value "computed_val" will be set by a return statement within the called function,}
{to be used in expressions of the form "a=f(b);" or "if(g(f()) > 3)..."}
return(*ec* ∧ *cond_call*)
end

{in case of a return statement}
if(NODE_TYPE(n) = *Return*) then begin
{consider the conditions potentially encountered while computing the returned-value}
$cond\_val = TRUE$
{set the "computed_val" parameter to the current value of the expression to be returned}
computed_val := GET_VALUE(RET_VAL(n),n_a, vs_g, vs_l, *path_leaves*, *cond_val*)
return(*cond_val*)
end

{in case of a VarVal expression, e.g. a symbolic variable location in memory}
if(NODE_TYPE(n) = $VarVal$) then begin
{generally, return the value part of the VarVal triplet as it's symbolic evaluation}
if (VAL(n) is undefined) then begin
{return the node n itself, will be encountered after dynamic allocation only}
computed_val := n
return(TRUE)
end else begin
computed_val := $VAL(n)$
return(TRUE)
end
end

{in case of a constant or symbolic expression, return the node itself (no need of further evaluation)}
if(NODE_TYPE(n) ∈ {*Const*, *Symbol*}) then begin
computed_val := n
return(TRUE)
end

```
{in case of the starting or ending nodes of a function Cfg}
if(NODE_TYPE(n) ∈ {Start, Stop}) then begin
      computed_val := n_a
      return(TRUE)
end

{in case of an unary expression}
if(NODE_TYPE(n) = Unary_exp) then begin
   {consider the conditions potentially encountered while computing the value}
   cond := TRUE
   if (OP(n) = *) then begin
      {obtain the address (the VarVal) corresponding to the child of n}
      computed_val := GET_VALUE(CHILD(n),n_a, vs_g, vs_l, path_leaves, cond)
   end
   else if (OP(n) = &) then
      computed_val := CHILD(n)
   else
      computed_val := n
   return(cond)
end

{in case of an binary expression}
if(NODE_TYPE(n) = Binary_exp) then begin
   {ec will accumulate all predicates encountered while computing node n}
   ec = TRUE
   if (OP(n) = .) then begin
      {compute the address of LEFT_CHILD(n).RIGHT_CHILD(n), to be re-evaluated as }
      {needed to obtain the current value}
      base := GET_ADDRESS(LEFT_CHILD(n),n_a, vs_g, vs_l, path_leaves, cond)
      ec := ec ∧ cond
      computed_val := GET_ADDRESS(RIGHT_CHILD(n),base, vs_g, vs_l, path_leaves, cond)
      ec := ec ∧ cond
   end
   else if (OP(n) = →) then begin
      {compute the address of the field accessed through the pointer (LEFT_CHILD(n)), }
      {i.e. the value of LEFT_CHILD(n)}
      base := GET_VALUE(LEFT_CHILD(n),n_a, vs_g, vs_l, path_leaves, cond)
      ec := ec ∧ cond
      computed_val := GET_ADDRESS(RIGHT_CHILD(n),base, vs_g, vs_l, path_leaves, cond)
      ec := ec ∧ cond
   end
   else if (OP(n) = [ ]) then begin
      {compute the address of the array element LEFT_CHILD(n)[RIGHT_CHILD(n)] }
      base := GET_ADDRESS(LEFT_CHILD(n),n_a, vs_g, vs_l, path_leaves, cond)
      ec := ec ∧ cond
      index_exp := GET_VALUE(rch,n_a, vs_g, vs_l, path_leaves, cond)
      ec := ec ∧ cond
      {check for symbolic indexing}
```

```
        if (NODE_TYPE(index_exp) = Symbol) then begin
            newindex := EXTERNALLY_OBTAINED_INDEX( ) {reads from a file}
            ec = ec ∧ (index_exp = newindex)
            index_exp := newindex
        end
        computed_val := GET_FIELD(base, index_exp)
    end else begin
        {Binary expression other than . → [ ]}
        lch := GET_VALUE(LEFT_CHILD(n),n_a, vs_g, vs_l, path_leaves, cond_lch)
        rch := GET_VALUE(RIGHT_CHILD(n),n_a, vs_g, vs_l, path_leaves, cond_rch)
        ec := ec ∧ cond_lch ∧ cond_rch
        {check for pointer arithmetic, e.g. if one side evaluates to a memory location}
        {while the other side is a constant, with operators + and -}
        if (NODE_TYPE(lch) = VarVal and NODE_TYPE(rch) = Const and OP(n) ∈ {+, −} )
        then begin
            newptr = INC(lch,rch)
            {create temporary VarVal that will be reevaluated later, yielding newptr}
            computed_val = < n_a, newptr, n_a >
        end
        {symmetrical case}
        if (NODE_TYPE(rch) = VarVal and NODE_TYPE(lch) = Const and OP(n) ∈ {+, −} )
        then begin
            newptr = INC(rch,lch)
            {create temporary VarVal that will be reevaluated later, yielding newptr}
            computed_val = < n_a, newptr, n_a >
        end
    end
    return(ec)
end

{in case of an identifier expression, e.g. variable name, field name or user function name}
if(NODE_TYPE(n) = Identifier) then begin
    {analyze elements of n}
    {note that functions IS_STRUCT_ATTRIBUTE, IS_GLOBAL_VARIABLE,}
    {IS_LOCAL_VARIABLE and IS_FUNCTION are not given }
    {but they basically map n into one of the categories described in section 4.1}
    if (IS_STRUCT_ATTRIBUTE(n)) then
        {compute the VarVal corresponding to that field}
        computed_val := GET_FIELD(base,VAR_ID(n))
    else if (IS_GLOBAL_VARIABLE(n)) then begin
        {compute the VarVal associated to that variable, or create it if inexistent}
        computed_val := vv ∈ vs_g | VAR_ID(VAR(vv))=VAR_ID(n)
        if (computed_val is inexistent) then begin
            computed_val := < n, n_a, n_a >
            vs_g := vs_g ∪ computed_val
        end
    end else if (IS_LOCAL_VARIABLE(n)) then begin
        {compute the VarVal associated to that variable, or create it if inexistent}
        computed_val := vv ∈ vs_l | VAR_ID(VAR(vv))=VAR_ID(n)
```

```
            if (computed_val is inexistent) then begin
               computed_val := < n, n_a, n_a >
               vs_l := vs_l ∪ computed_val
            end
         end else if (IS_FUNCTION(n)) then
            {no computation needed}
            computed_val := n_a
         return(TRUE)
      end

      {in case of an system function call expression/statement}
      if(NODE_TYPE(n) = SystemFctCall) then begin
      {some basic C functions have been defined, for example}
         if C_FUNCTION_NAME(n) = malloc then begin
            {create a new VarVal in the heap (global-level)}
            computed_val := < n_a, n_a, n_a >
            vs_g := vs_g ∪ computed_val
         end
         else if C_FUNCTION_NAME(n) = getch then begin
            {create and return a new symbolic value s with unique identification (name)}
            computed_val := s | s ∈ Symbol
         end
         else if C_FUNCTION_NAME(n) = scanf then begin
            {associate new symbolic values for each of the input variables}
            ∀ input variables v begin
               lch := GET_ADDRESS(v,n_a, vs_g, vs_l, path_leaves, n_a)
               SET_VAL(lch, s) | s ∈ Symbols with s unique
            end
            computed_val := n_a
         end
         return(TRUE)
      end
   end function



Function INC(in e : Expression; offset : N):Expression;
begin
   {INC increments not only by a value of 1, but by a value of offset (positive or negative)}
   if (NODE_TYPE(e) = VarVal) then begin
      {if e is a pointer, compute element at offset distance from e}
      res := e' | e, e' ∈ fields ∧ VAR_ID(VAR(e')) = VAR_ID(VAR(e)) + offset
   else
      {arithmetic incrementation}
      res := e + offset
   return(res)
end function
```

Function **SET_VAL**(in VAR $vv : VarVal$; $new\_val : Expression$)
begin
  {*modify the value section in the triplet received*}
  $id' :=$ VAR(vv)
  $fields' :=$ FIELDS(vv)
  $vv := < id', new\_val, fields' >$
end function

Function **GET_FIELD**(in $vv : VarVal$; $index : Const$):$VarVal$
begin
  {*computes and returns field index within VarVal vv or create it if inexistent*}
  $res := vv' \in FIELDS(vv) \mid index = VAR\_ID(VAR(vv'))$
  if (res is inexistent) then begin
    $res := < n, n_a, n_a >$
    $FIELDS(vv) := FIELDS(vv) \cup res$
  end
  return(res)
end function

Function **ROOT**(in $V : 2^{V_{ET}}$; $E : 2^{E_{ET}}$):V_{ET}
begin
  {*computes and returns the root node of a given tree* $T = (V, E)$}
  $v := w \in V \mid \nexists < w', w >\in E$
  return(v)
end function

Function **FIRST**(in $< v_1, v_2, ..., v_n >\mid v_i \in V_{ET}$):$V_{ET}$
begin
  {*returns the first element of a received list*}
  return(v_1)
end function

Function **GET_ADDRESS**
(in $n : V_{CFG}$; $base : VarVal$; $vs_g, vs_l : State$; $VAR\ path\_leaves : 2^{VET}$; $VAR\ cond : Expression$):
$Expression$
begin
  {*unique evaluation of n, which will give a reference*}

{*to the VarVal associated to n in memory*}
cond := EVAL_NODE($n, n_a, base, vs_g, vs_l, path\_leaves, adress$)
return(*adress*)
end function

Function GET_VALUE
(in $n : V_{CFG}; base : VarVal; vs_g, vs_l : State; VAR\ path\_leaves : 2^{V_{ST}}; VAR\ cond : Expression$):
Expression
begin
{*evaluation of the address (the VarVal) of n, which will*}
{*result in the current value of that VarVal*}
adress = GET_ADDRESS($n, obj\_from, vs_g, vs_l, path\_leaves, cond$)
cond := *cond* $\wedge$ EVAL_NODE($adress, n_a, base, vs_g, vs_l, path\_leaves, value$)
return(*value*)
end function

C LINEAR OPTIMIZATION PROBLEM

Once a logical expression has been computed after having symbolically executed a given partial execution tree to reach an edge in the program, transformation of this problem into a set of linear constraints is required in order for the problem to be submitted to an optimization tool which will try to associate values to symbols of the equation that will validate the problem.

Function $lp : EC\_EXPR \rightarrow LC$ which computes the set of linear constraints associated to logical expression e can be obtained with:

$$lp(e) = lc(e, \{\})$$

To define function lc, we need to state that expression e follows the following rules:

e is an expression that can be composed of

$$expression\ AND\ expression$$

$$expression\ OR\ expression$$

$$variable\ operator\ constant$$

$$(with\ operator \in Operator\ previously\ defined)$$

In the context where e is of the form *variable operator constant*, we define the following functions to gain access to the constituents of e:

$$VAR(e)\ is\ the\ variable\ associated\ with\ e$$

$$CST(e)\ is\ the\ constant\ associated\ with\ e$$

Function $lc : EXPR \times 01VarsSet \rightarrow LC$, where $01VarsSet$ is a set of 01 variables, is defined as follows:

$$lc(e, OrVars) =$$

$$\begin{cases} lc(RIGHT\_CHILD(e), OrVars) \\ \cup \\ lc(LEFT\_CHILD(e), OrVars) & \text{if OP(e) = AND} \\ \\ lc(RIGHT\_CHILD(e), OrVars \cup \{z_1\}) \\ \cup \\ lc(LEFT\_CHILD(e), OrVars \cup \{z_2\}) \\ \cup \\ z_1 + z_2 <= 1 \\ (\textit{where } z_1 \textit{ and } z_2 \textit{ are two new 01 variables}) & \text{if OP(e) = OR} \\ \\ VAR(e) < -M \times z_i,\ \forall z_i \in OrVars > <= CST(e) \\ \cup \\ VAR(e) < +M \times z_i,\ \forall z_i \in OrVars > >= CST(e) & \text{if OP(e) = EQ} \\ \\ VAR(e) + M \times z < -M \times z_i,\ \forall z_i \in OrVars > <= CST(e) - 1 + M \\ \cup \\ VAR(e) + M \times z < +M \times z_i,\ \forall z_i \in OrVars > >= CST(e) + 1 \\ (\textit{where } z \textit{ is a new 01 variable}) & \text{if OP(e) = NEQ} \\ \\ VAR(e) < -M \times z_i,\ \forall z_i \in OrVars > <= CST(e) & \text{if OP(e) = LEQ} \\ \\ VAR(e) < -M \times z_i,\ \forall z_i \in OrVars > <= CST(e) - 1 & \text{if OP(e) = LT} \\ \\ VAR(e) < +M \times z_i,\ \forall z_i \in OrVars > >= CST(e) & \text{if OP(e) = GEQ} \\ \\ VAR(e) < +M \times z_i,\ \forall z_i \in OrVars > >= CST(e) + 1 & \text{if OP(e) = GT} \end{cases}$$

where M is a proper large enough constant (1000 was used for our experiments).

Note: boolean unary operator NOT is not present in the above algorithm, because expressions containing NOT operators are transformed into their logical equivalent, for example: $NOT(X < 5)$ is transformed into $X >= 5$. De Morgan's laws were also used in connection with logical conjunctions and disjunctions in expressions to simplify the NOT operator.

Annexe B

"Test Case Generation for Object-Oriented Software Using Symbolic Execution"

Article écrit par S. Lapierre, E. Merlo, G. Savard et G. Antoniol et soumis pour publication.

Test Case Generation for Object-Oriented Software Using Symbolic Execution

S. Lapierre[1], E. Merlo[1], G. Savard[1], G. Antoniol[2]

[1] *GEGI, École Polytechnique*, C.P. 6079, Succ. Centre Ville. Montréal, Québec, Canada.
lapierre@casi.polymtl.ca, merlo@rgl.polymtl.ca, gilles@crt.umontreal.ca

[2] *IRST-Istituto per la Ricerca Scientifica e Technologica*, I-38050 Povo (Trento), Italy.
antoniol@irst.itc.it

ABSTRACT

This paper presents an approach to automated software test data generation for branch coverage of C++ programs. The approach is based, in part, on symbolic execution of program paths. Symbolic execution of object-oriented features has been performed and is presented in details, which constitutes a major contribution of this article. Correctness of the approach is shown by preliminary results, which are very encouraging and demonstrate that object-oriented code can be treated by symbolic execution and that a high degree of program branch coverage can be achieved using this technique.

1 INTRODUCTION

The object-oriented (OO) paradigm is rapidly gaining acceptance in the software industry [3]. While the increased power of object-oriented languages create new opportunities for error, reusability of object-oriented components brings forth an increased need for highly reliable software, accentuating the extreme importance of OO software testing [12, 14].

Characteristics of object-oriented software, mainly it's event-driven nature, imply modifications to the conventional approaches to testing, emphasizing the need for new integration testing principles [8, 10]. Object state and behavior are new realities that need to be studied for an adequate OO testing approach to be proposed, opening interesting research avenues for functional testing research. Conventional structural testing approaches, however, need not be discarded and still have an important role to play in OO software testing. Though not necessarily complete for analyzing OO software, combining them with more recent testing approaches will lead to a very high degree of reliability in the tested software. The goal and approach of unit testing, however, remain similar through the paradigm change.

Generally speaking, software testing aims at identifying and removing software defects in the analyzed programs. Unit testing deals with software units, a unit being defined, in this paper, as a function or a method in the program, while integration testing analyzes multiple units interacting with each other, to the point where every unit is present and active in the test environment, in which case integration testing becomes system testing. While doing unit and integration testing, one can adopt a functional or a structural approach to the tests performed. In functional (black-box) testing, the external behavior of the tested software is monitored, e.g. the outputs of the program are analyzed for conformity with respect to the expected outputs, given the input values used while testing. Structural (white-box) testing, however, is interested by the internal structure of the software and usually aims at achieving coverage of a given criterion, for example covering all program statements at least once, or program branches, or definition-use chains. The testing goal is thus centered around generating input data that will enable to exercise all the program statements, branches, definition-use chains, and so on, depending on the coverage criterion to be achieved.

Our research is interested in automating input data generation for structural testing, and more precisely to achieve branch coverage of a tested program. Automated test data generation, in the past, has mainly been an area of research limited to procedural language softwares. Traditional approaches to achieve test data generation for procedural software will be presented in section 2, followed by a general description of our specific approach and the presentation of the adaptations required in order to automatically generate input test data in section 3. Following that section, the preliminary results of our research will be presented and discussed (section 4). The paper will end with ideas for future research (section 5) and the conclusion of this work (section 6).

2 SOFTWARE TESTING AUTOMATION

As stated above, white-box testing relies on the internal structure of the program to be tested. To achieve test data generation based on this approach, two main research directions have been followed, the first one being based on dynamic analysis and the second one on symbolic execution. It is to be noted that these ap-

proaches have been essentially applied, in past systems, to procedural language software.

2.1 Dynamic generation of test data

Test data generation based on a dynamic approach [9] relies on an actual execution of the program to be tested. During this execution, the program is externally monitored and modifications are applied to it's input data in order to force an execution through a given program path. Input data modifications are performed using successive optimizations of the program variables influencing a function built with the accumulated conditional predicates encountered along the execution path.

2.2 Symbolic-execution based generation of test data

Test data generation systems based on symbolic execution usually implies 3 distinct modules, which are:

- path generator
- symbolic executor / constraint generator
- inequality solver

In the path generation phase, the program's structure is statically analyzed and a given path selection strategy is applied in order to determine specific paths that will have to be processed to achieve coverage of a given testing criterion.

Symbolic execution is then performed on the resulting paths, one at a time. Symbolic execution could be described as a simulation of a real execution on a given path, with all input values left unknown and being replaced by symbolic values. Conditional predicates encountered along the execution path are accumulated into a Path Condition (PC) containing symbolic expressions of the variables composing each such conditional predicate. As a result, the PC contains a conjunction of all the distinct predicates encountered, expressed as a function of symbolic expressions representing inputs.

An inequality solver module is given the task of analyzing the resulting PC in order to deduce some symbolic values that will render it valid, thus finding some actual input values that will force a real execution of the program through the path of interest.

3 GENERAL DESCRIPTION OF OUR APPROACH FOR AUTOMATIC TEST CASE GENERATION

3.1 General Theoretic Approach

In order to automatically generate input test data to be used as test cases for program branch coverage, our approach uses different concepts, such as unconstrained edges in a program graph, execution trees and symbolic execution.

A program to be analyzed is first of all parsed in order to generate intermediate representations of the program components, called program control flow graphs (CFG). For a given program, each sub-program (function or method) is represented by such a CFG, which is a graph where each node represents a statement in the sub-program and each edge between nodes represents a possible flow of control between statements. To illustrate this concept, the $CFGs$ for the 2 sub-programs of *program0* (shown in figure 1) are presented in figure 2.

```
1 :   class A {
2 :     public:
3 :       int a ;
4 :       void f(void) {
5 :         while (a < 10)
6 :           if (a < 5)
7 :             a++ ;
8 :       }
9 :   } ;

11 :  main() {
12 :    A var ;
13 :    cin >> var.a ;
14 :    var.f() ;
15 :  }
```

Figure 1: C++ Code for program0

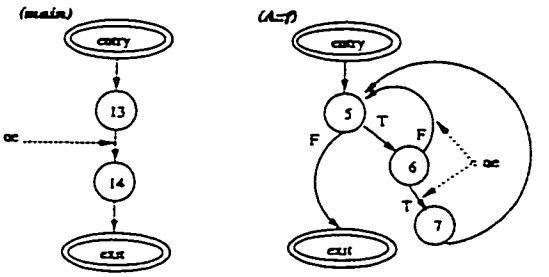


Figure 2: *$CFGs$ for sub-programs of program0, where node labels represent statement line number in the source program.*

The $CFGs$ obtained from parsing are afterwards analyzed in order to produce unconstrained edges information. Unconstrained edges computation is a technique described in [1, 2, 6] which basically enables us to deduce a minimal set of edges in the $CFGs$, corresponding directly to branches in the program, with the following property: if an input data set forces execution to reach all the edges in the unconstrained edges set, then it is

certain that all program paths will be covered. Unconstrained edges computation is based on concepts of dominance and post-dominance between elements in a graph; in other words, if execution of a graph element e_1 implies execution of another graph element e_2, then the implied element e_2 need not be directly covered by a test case, because covering e_1 will necessarily imply coverage of e_2. See [1, 2, 6, 11] for further details on unconstrained edges description and computation. As an example, the set of unconstrained edges computed for *program0* is composed of the following CFG edges: $\{< 6,5 >, < 6,7 >, < 13,14 >\}$ which are identified on the $CFGs$ of figure 2 by dotted-lined-arrows.

Having computed the set of unconstrained edges UE, the new goal of test data generation for program branch coverage is to produce a test case that will enable coverage of each branch corresponding to edges of UE.

Let e_{UE} be such an edge of UE for analyzed program P and b_{UE} be it's associated branch in P. In our approach, generating a test case to cover b_{UE} resumes to finding input values, to be used while executing P, that will force control to follow any sub-path *sub_path* in P ending at b_{UE}, with the restriction that any node $n \in$ *sub_path* must not be traversed more than k times while traversing *sub_path*.

Computing such a test case involves generating a partial execution tree to reach edge e_{UE}. Execution trees were introduced in [7, 13] and used in automated testing in [11]. Informally, a partial execution tree for branch b_{UE} is a united collection of sub-paths ending at branch b_{UE}, where each statement in the sub-paths is encountered no more than a fixed number of times (k).

To illustrate this principle, the partial execution tree computed to reach unconstrained edge $< 6,7 >$ of *program0*, considering an iteration limit of $k = 2$, is given in figure 3. Note that the partial execution tree reaching edge $< 6,7 >$ within $k = 2$ iterations is given in the context of integration testing, e.g. it's root corresponds to the first statement of the *main* program even though edge $< 6,7 >$ is part of method $A :: f$. The complete *sub_path* is therefore given from the beginning of *main* and ending at edge $< 6,7 >$. In this context, partial execution tree node 14 is identified by a dotted circle because the statement at line 14 corresponds to the method call $A :: f$ which is not fully executed but is expanded through it's children nodes in the tree, e.g. the nodes corresponding to statements of $A :: f$.

Symbolic execution is then performed on each sub-path included in this tree. Symbolic execution can be described as a simulation of an execution, along a given program sub-path, where each statement's actions are simulated. A special characteristic of symbolic execution is that input statements introduce unknown data

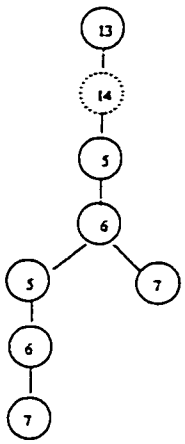


Figure 3: *partial execution tree generated to reach unconstrained edge* $< 6,7 >$, *considering an iteration limit of* $k = 2$.

values, which are left unknown and are termed as "symbols". Conditional statements that are encountered along the sub-path are evaluated and their predicates are accumulated into a pc (path condition) which is a logical expression, function of input symbols, which has to be validated for the sub-path to be traversed. In our system, symbolic execution of the sub-paths which compose a partial execution tree yields a tree decorated with the predicate conditions encountered while performing symbolic execution. This decorated tree is thus composed of different symbolically conditioned sub-paths and test data generation is interested in finding input data to force execution along any one of these sub-paths. This corresponds to finding values for symbols which will validate any one of the *pcs* of the sub-paths in the execution tree.

The *pcs* are transformed into a logical symbolic expression with the particularity that a branching in the tree introduces a disjunction in the expression "$\vee$" (one sub-path OR the other is to be validated) while non-branching nodes introduce a conjunction "$\wedge$". Formulating the conditions to reach a given node with such disjunctions and conjunctions produces a partial branch coverability condition ($PBCC$), which enables the representation of multiple alternative path conditions within a single condition, with the distinct advantage of bypassing most problems related to infeasible paths and non-linear path conditions, given that at least one linear feasible path exist in the partial execution tree analyzed. The logical condition generated after symbolically executing the partial execution tree of figure 3 is given in figure 4.

$$VAR\_ID(id) = \begin{cases} > 0 & \text{if } id \text{ is a variable or struct/class attribute, } var\_id \text{ then is the unique integer associated to that variable/attribute} \\ < 1 & \text{if } id \text{ is a class} \\ 0 & \text{otherwise} \end{cases}$$

$$TYPE\_ID : IdentifierSpace \rightarrow type\_id, \text{ with}$$

$$TYPE\_ID(id) = \begin{cases} > 1 & \text{if } id \text{ is a variable(of OO type) or class attribute(of OO type), } type\_id \text{ then is the unique integer associated to the type-class of } id \\ 0 & \text{otherwise} \end{cases}$$

Examples of the constraints used for identifier representations are given in table 1.

Identifier category	class_id	func_id	var_id	type_id
user function/main	0	> 0	0	0
class attribute	> 1	0	> 0	0 or > 1
struct. attribute	1	0	> 0	0 or > 1
global variable	0	0	> 0	0 or > 1
local var./parameter	0	> 0	> 0	0 or > 1
method	> 1	> 0	0	0
class	> 1	0	0	0

Table 1: *Examples of identifier representation.*

3.2.3 Inheritance Inheritance relationships are preserved and stored in the intermediate representation of the analyzed program in the form of trees where the roots of the trees represent classes that do not inherit from any other class. This information is particularly needed when calling virtual methods through object pointers, as illustrated in example at section 3.4.2, because the actual method called will be dependent upon the type of the pointed-to object, and hierarchical relationships will have to be consulted for the correct method to be called.

Multiple inheritance is not currently treated by our analyzer.

3.2.4 Polymorphism Polymorphic functions or methods information is available while parsing the analyzed program. Two sub-programs identifiers with identical names but with different number or type of parameters will be represented by different quadruples in our system, eliminating any possibility of confusion.

3.2.5 Pointers Symbolic execution being performed on a given program path at a time, pointer operations can be directly mapped into memory manipulations of the symbolic internal variables within the symbolic executor. The precise algorithms used for symbolic pointer manipulations are given in appendix A and are illustrated by examples 1 and 2 in sections 3.4.2 and 3.4.3. Refer to [4, 5] for further details.

3.2.6 Method invocation When performing symbolic execution of a given program, encountering a function/method call is treated by symbolically executing a given path through the called function/method.

Determining the correct method to be invoqued in an ordinary method call such as $o.f(...)$ is simply done by evaluating the type class of the calling object and searching for the corresponding class method in the class hierarchy tree.

An essential information to propagate while performing that call concerns the actual object from which a method has been called, e.g. the object o in $o.f(...)$. The reason for this requirement is the following: attributes accessed while symbolically executing method f will have to be treated as attributes of object o. In order to do so, the base object used in any method call is always used as parameter *obj_from* in our symbolic execution algorithms (see appendix A).

3.2.7 Virtual method invocation and function pointers When in presence of function pointers or object pointers when invoking virtual methods, the difficulty lies in determining the correct function/method to be called.

Function pointers are quite straightforward to process because of the context of symbolic execution, e.g. because symbolic execution is done on a given program path and that values of variables, function pointers included, are unambiguously known at each step of the execution along this path. Accessing the CFG pointed-to by a function pointer, at a given point of the program, is done by simply evaluating that pointer variable.

Determining the actual method called when calling a virtual method m, in presence of an object pointer p, is also quite straightforward. Along a given execution path, it is always a simple matter to evaluate the type of the actual object o which is pointed by p. Let us state, as a reminder of section 3.2.2, that object variables are typed in our test data generation system. Invoking m through pointer p corresponds to invoking m through object o. The correct version of m in a hierarchical class tree is found by looking if a method m has been declared for the class of object o, and if not, by searching for a method m in the parent classes of the class of object o, taking the version of m found in the closest parent class during the search.

After the correct function/method has been identified, symbolic execution is performed as for any ordinary function/method call, as explained in section 3.2.6 Example 1 (section 3.4.2) illustrates symbolic execution of virtual method calls through an object pointer.

3.2.8 Static attributes Static attributes are attributes associated to a whole class and not to each and every declared object of that class. In our system, if an attribute is identified as being static, then it is associated to a global variable associated to the corresponding class. As can be seen in the example of section 3.4.3, if a static attribute $A :: glob$ is used, then a global variable will be created to represent class A, and a field (attribute) *glob* will be created for that variable. In that way, every access to $A :: glob$ through different object of class A or inheriting from A will access the field *glob* of global variable A.

3.3 Symbolic execution of object-oriented programs

Symbolic execution has been extended to deal with the aspects of object-oriented programs, as mentioned above. The algorithms for symbolical execution of object-oriented code are given in appendix A.

3.4 Examples of symbolic execution

Two examples have been chosen to help understand our technique for symbolically executing object-oriented code. Example 1 has been chosen because it involves a call to a redefined virtual method through an object pointer. The correct method therefore has to be determined and invoqued, which implies the need of evaluating the precise class of the pointed-to object and looking-up for the virtual method associated to that class in the method/class hierarchy. Example 2 is interesting because it contains manipulations of static data class attributes, which call for special considerations from our symbolic executor, e.g. it must be considered that a unique attribute *class :: attribute* is to be manipulated whenever an object of class *class* or it's descendants is to perform an operation on *attribute*. Before presenting in details the symbolic execution examples, a note has to be given about the terminology used for test data representation within our system.

3.4.1 Data values representation Current values of variables through symbolic execution are represented as a *State*, with $State = \{VarVal\}$, a *State* being a collection of $VarVal$ which are associations between variables and their current values or fields (structure/object attributes or array elements). A $VarVal$ is a triplet $< var, val, fields >$ with

$$var \in IdentifierSpace$$

$$val \in Expression(current\ value\ of\ var)$$

$$fields \in State$$

3.4.2 Example 1 Let's take *program1* as an example (see figure 10). Every path within *program1* will inevitably follow the subpath composed of the statements at lines 16→17→35→36→37→38→18.

Let us examine the actions taken by the symbolic executor while analyzing this sequence of statements. To help the reader better understand the analysis, graphic representations of the symbolic executor variables state will be given throughout this and the following examples.

At line 16 ($ptr = \&var$;), a pointer assignation is performed. In the executor, the memory location of the $VarVal$ associated to program variable *var* will be given as the value of the $VarVal$ representing program variable *ptr*. Internal state of the symbolic executor after evaluating line 16 can be viewed in figure 6.

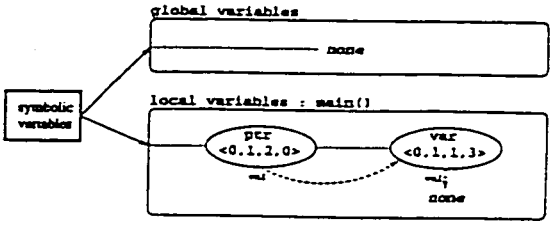


Figure 6: *Symbolic executor's variables states after evaluating statement at line 16 of program1.*

Line 17 ($ptr \rightarrow set(5)$;) is a method call where the access path is given by $ptr \rightarrow set$. Analysis of this access path will be performed as follows:

1. evaluation of the binary operator →, which will recursively evaluate the value of it's left component and of it's right component.
2. evaluation of the left component, yielding $VarVal$ *var*, denoted $VarVal(var)$.
3. evaluation of the right component with $obj\_from = VarVal(var)$, e.g. in the context of $VarVal(var)$ as the calling object, will result in determining a method, *set*, which is defined in the base class A and not redefined elsewhere.

Following the evaluation of the access path, the executor will proceed with the evaluation of the argument of the method call, in this case a constant argument of value 5. The executor will then create a local variable space for the called method *set*, with parameter variable *i* being initialized to 5, followed by a recursive symbolic execution performed within method *set*.

Internal state of the symbolic executor while evaluating line 17 ($ptr \rightarrow set(5)$;), just before evaluating statements within method *set*, can be viewed in figure 7.

Evaluation of line 36 ($scanf("\%d", \&a)$;) resumes to evaluating the call to library function *scanf*, evaluation

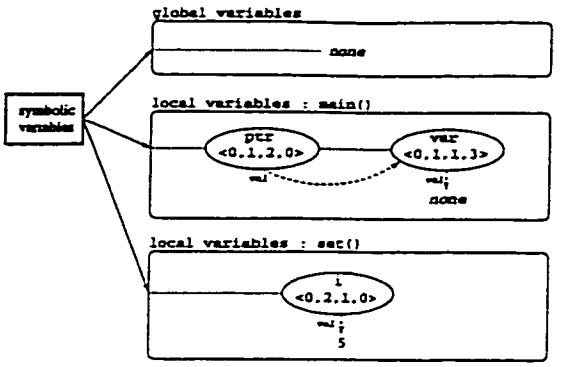


Figure 7: *Symbolic executor's variables states while evaluating statement at line 17 of program1, just before beginning evaluation within method set.*

which is predefined within our symbolic executor. This evaluation will follow these steps:

1. assignment of symbolic values to the 2..n arguments of the call to $scanf(1, 2, .., n)$ function.
2. for argument $\&a$ of $scanf$, the executor will detect that a is a class attribute and will consequently manipulate field a of the object with which *set* was invoked, e.g. $VarVal(var)$.
3. symbolic value $X1$ will be assigned to field a of internal variable $VarVal(var)$.

Internal state of the symbolic executor after evaluating line 36 is given in figure 8.

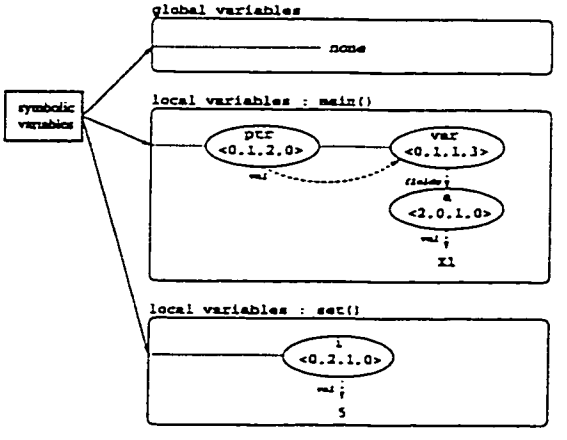


Figure 8: *Symbolic executor's variables states after evaluating statement at line 36 of program1.*

While evaluating line 37 ($b = i$;), attribute b of $VarVal(var)$ will be given the current value of variable i, in this case constant value 5. Internal state of the symbolic executor after evaluating line 37 can be viewed in figure 9.

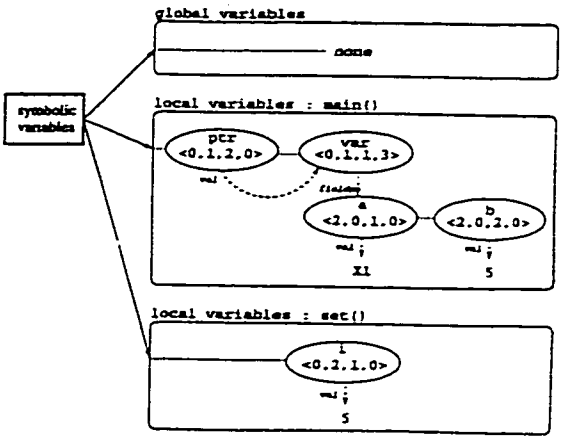


Figure 9: *Symbolic executor's variables states after evaluating statement at line 37 of program1.*

Evaluating line 38 ($f()$;) will be as follows:

1. evaluate the access path of the call, in this case f. f being a class method, it will be defined in this case as quadruple $< 2, 1, 0, 0 >$, e.g. first method of class number 2 (class A). f is defined, by default, as a method of class A because *set* is also of class A.
2. the executor will check if f is defined as being virtual, which will be true in this particular example. f being virtual, a search will be performed in the class hierarchy available to the executor, searching for a method with same method number as method f (i.e. 1 in $< 2, 1, 0, 0 >$) but being a member of the closest class possible to the actual class of the context object, *var* in this case, an object of class 3 (B). This search will yield method $f = < 3, 1, 0, 0 >$, which corresponds to method $B :: f$ and which is the correct method to be called here.

Evaluation of the following statements follows the rules above mentioned or the basic rules of symbolic execution.

3.4.3 Example 2 Given *program2* (see figure 14), every execution will necessarily traverse statements at lines 17→18→36→37→38→39→40.

Evaluation of statement at line 17 ($ptr = \&var$;) is the same as in the previous example, except that variable

ptr is now a global variable and therefore is in a different variable space. Evaluation of line 17 yields internal state of the symbolic executor given in figure 11.

Evaluating statement at line 18 ($ptr \rightarrow set(\ )$;) is essentially the same as already explained earlier (see execution of statement at line 17 of *program*1), with the exception that no argument is passed in the method call. Evaluating statements at lines 37 ($scanf("\%d", \&a)$;) and 38 ($scanf("\%d", \&b)$;) of *program*2 also are similar to the evaluation of line 36 ($scanf("\%d", \&a)$;) of *program*1. Internal state of the symbolic executor after evaluating line 38 can be viewed in figure 12.

Execution of statement at line 39 ($A :: glob = a+b$;) is interesting because it involves a static attribute of class A. While analyzing line 39, our executor will evaluate the left-hand-side of the assignation, which corresponds to $A :: glob$, in the following steps:

1. evaluation of the binary operator ::, where an evaluation will be performed of the left component of the operator, which we expect to be a class designation, as A in this example.

2. Evaluation of A will result in the quadruple $< 2, 0, 0, 0 >$ designating class A.

3. Evaluation of :: will be continued and an evaluation of it's right component will be performed with $obj\_from =< 0, 0, -2, 2 >$, e.g. a new global variable will be created which will have as variable number -1*(class number of A) and which will be of type A, e.g. 2.

4. Field *glob* of this new global variable will then be assigned the evaluated value of the right-hand-side of the assignation, $X1 + X2$ in this case.

Internal state of the symbolic executor after evaluating line 39 can be viewed in figure 13.

The evaluation of the statement at line 40 ($f(\ )$;) of *program*2 will first analyze the access path of the method called, in this case f. Evaluation of this access path will yield method $A :: f$, because that is the method specified in *set*, e.g. by default, the class appartenance chosen for f in *set* will be the same as the one for *set* itself, A in this example. The obtained method f is not defined as being virtual, so no further search is conducted and method $A :: f$ is invoked at line 40 of *program*2.

4 PRELIMINARY RESULTS AND DISCUSSION

The concepts presented in this paper have been implemented into a tool called *TAO* [1] which is coded in approximately 20 Kloc, in C++, and compiled under g++ compiler on a PentiumPro 166Mhz processor with 64Mb RAM and running Linux operating system.

At the present stage of this research, 5 C++ language programs have been analyzed through our test data generator. The five programs can be termed as academic and were designed to test specific features of object-oriented programs, as presented in table 2.

Given the analyzed programs, *TAO* was able to correctly generate test data to exercise all feasible paths available (some of the programs contained infeasible paths, for example, all branches of method $A :: f$ cannot be traversed in program 1 of figure 10 because no call to that method is performed in the program, when performing integration testing).

TAO's symbolic executor successfully detected and treated all the above-mentioned characteristics of object-oriented programming.

Even though the programs number and sizes are very limited, preliminary results indicate that the approach correctly analyzes object-oriented features and adequately generates input data to cover the program's branches.

Experiments on procedural C-language programs were done and presented in [11]. In summary, 12 programs containing a total of 41 units were analyzed in that experiment, using techniques similar to the ones presented in this paper. Automatic unit test data generation was performed on these units, resulting in a successful 98.5% coverage of those unit's unconstrained edges. The total time required to attain this coverage was less than 80 seconds.

Even though the technique presented in the present paper is object-oriented, the test data generation concepts involved when dealing with procedural [11] or object-oriented programs are similar. Is it our opinion, based on the promising results of [11], that the extension of the test data generation method to object-oriented programs is worthwhile to be further investigated. Present C++ experiments confirm also the feasibility of the approach, although more data are needed before stating general conclusions.

[1] *TAO* is a French acronym which stands for *Tests Assistés par Ordinateur*

Program	number of lines of code	characteristic
program 1	43	call of a virtual, redefined, method through an object pointer (variant)
program 2	44	access to a static class attribute
program 3	34	simple method calls and access to class attributes within the methods
program 4	42	call of a non-virtual, redefined, method through an object pointer
program 5	42	call of a virtual, redefined, method through an object pointer

Table 2: *Number of lines of code and characteristics for the 5 analyzed programs.*

5 FUTURE RESEARCH

Two major areas of future research can be identified. The first area of research is related to treatment of object-oriented features which are not currently treated by our system. Such features include analysis of multiple inheritance and template use in C++ code. Another C++ related point of research is to correctly analyze constructor calls, especially when in presence of multiple global objects where the order of constructor calls is of importance. More experiments on inheritance and other OO related issues should be performed before drawing general conclusions.

The second area of research is related to improvements that could be made regarding test data generation, in particular:

- explore possibilities for processing non-linear equations
- improve the analysis done when encountering symbolic indexing
- integrate data-flow testing criterion to conduct test data generation

6 CONCLUSION

An approach has been presented to automatically generate test data for branch coverage of C++ software. The approach is based on unconstrained edges in a program control flow graph, on the use of execution trees and on symbolic execution of program paths in order to generate and finally solve equations for alternative paths reaching a minimal set of edges enabling full branch coverage of the program. Special attention has been given to symbolic execution of object-oriented features such as virtual method called from object pointers and static attribute access.

The presented methods have been successfully implemented into an automatic test data generation tool and preliminary experiments show their accuracy. Further research and experiments need to be performed in order to consolidate the basis of the method.

REFERENCES

[1] H. Agrawal. "Dominators, Super Blocks, and Program Coverage", *POPL'94:Symp. on Principles of Programming Languages*, 1994, pp.25–34.

[2] A. Bertolino and M. Marré. "Automatic generation of path covers based on the control flow analysis of computer programs", *IEEE Trans. Soft. Eng.*, dec. 1994, pp.885–898.

[3] G. Booch. "Object-oriented development", *IEEE Trans. Soft. Eng.*, feb. 1986, pp.211–221.

[4] A. Cimitile, A. De Lucia and M. Munro. "Qualifying reusable functions using symbolic execution", *IEEE proc. Second Working Conf. on Reverse Eng.*, july 1995, pp.178–187.

[5] A. Cimitile, A. De Lucia and M. Munro. "Identifying reusable functions using specification driven program slicing: a case study", *IEEE proc. Int. Conf. on Soft. Maintenance*", oct. 1995, pp.124–133.

[6] T. Chusho. "Test data selection and quality estimation based on the concept of essential branches for path testing", *IEEE Trans. Soft. Eng.*, may 1987, pp.509–517.

[7] R. Jasper, M. Brennan, K. Williamson and B. Currier. "Test data generation and feasible path analysis", *Int. symp. on soft. testing and analysis*, 1994, pp.95–107.

[8] P. C. Jorgensen and C. Erickson. "Object-oriented integration testing", *communications of the ACM*, sept. 1994, pp.30–38.

[9] B. Korel. "Automated software test data generation", *IEEE Trans. Soft. Eng.*, aug. 1990, pp.870–879.

[10] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, Y.-S. Kim and Y.-K. Song. "Developing an object-oriented software testing and maintenance environment", *Comm. of the ACM*, oct. 1995, pp.75–87.

[11] S. Lapierre, E. Merlo, G. Savard, G. Antoniol, R. Fiutem and P. Tonella. "Description of *TAO* test data generation system", *École Polytechnique technical report*,EPM/RT-97/25, aug. 1997.

[12] M. Lejter, S. Meyers and S. P. Reiss. "Support for maintaining object-oriented programs", *IEEE Trans. Soft. Eng.*, dec. 1992, pp.1045–1052.

[13] T. E. Lindquist and J. R. Jenkins. "Test-case generation with IOGen", *IEEE Soft.*, jan. 1988, pp.72–79.

[14] N. Wilde and R. Huitt. "Maintenance support for object-oriented programs", *IEEE Trans. Soft. Eng.*, dec. 1992, pp.1038–1044.

[15] D. F. Yates and N. Malevris. "Reducing the effects of infeasible paths in branch testing", *ACM SIGSOFT Soft. Eng. Notes*, 1989, pp.48–54.

```
1 :   class A {
2 :     public:
3 :       int a,b ;
4 :       virtual void f(void) ; // note : f is virtual
5 :       void set(int) ;
6 :   } ;

8 :   class B : public A {
9 :     public:
10 :      void f(void) ;
11 :  } ;

13 :  main() {
14 :    B var ;
15 :    A* ptr ; // pointer to base class
16 :    ptr = &var ;
17 :    ptr→set(5) ;
18 :    ptr→f() ;
19 :  }

21 :  void A::f(void) {
22 :    if (a > b)
23 :      b++ ;
24 :    else
25 :      a++ ;
26 :  }

28 :  void B::f(void) {
29 :    if (a > b)
30 :      a++ ;
31 :    else
32 :      b++ ;
33 :  }

35 :  void A::set(int i) {
36 :    scanf("%d", &a) ;
37 :    b = i ;
38 :    f() ;
39 :  }
```

Figure 10: C++ Code for program1

A SYMBOLIC EXECUTION OF OBJECT-ORIENTED CODE

Function EVAL_NODE is the function defined in our system to perform symbolic execution. It is designed to symbolically execute a given node, and is defined for all possible nodes in the system, whether they be statement or expression nodes.

Here is an explanation of the main features of EVAL_NODE function. The function takes 7 parameters, which are:

- n : the node to be symbolically executed
- o : (if applicable) o is the destination node of an edge $< n, o >$ for which we wish to compute an edge-condition.
- *obj_from* : when dealing with objects, *obj_from* gives a reference to the $VarVal$ corresponding to the object within which actions are performed, e.g. attributes are modified or a method is called, for example the $VarVal$ associated to a when performing $a.g(2)$.
- vs_g and vs_l : the sets of global-level and local-level variables currently computed and accessible (for local-level).
- *path_leaves* : vector of nodes which correspond to the execution tree leaves of the functions which will be called in order to execute node n. *path_leaves* can be empty if n doesn't perform a function call, or can be composed of numerous execution tree leaves if n performs numerous calls, for example if n is an assignation statement resembling $x = f(2 + g(h(1) * 3))$, which directly implies 3 function calls in the order h, g then f; in this example, a path will be identified within h, g and f and the leaf of that path in the corresponding execution tree will be associated to *path_leaves* for node v.
- *computed_val* : is a parameter passed by address and which will be used to return the computed value for the current node n. For example, if n is the binary expression $2 + 3$, parameter *computed_val* will be used to return the computed value 5.

The function EVAL_NODE is also defined to return an expression. This expression corresponds to the logical condition computed in order to execute node v. It has to be noted that this expression takes into account all the conditions encountered while evaluating node v. For example, if v is the assignation statement $x = f(2+g(h(1)*3))$, the conditional expression computed for node v will be a conjunction of the conditions accumulated while symbolically executing the paths ending at the nodes of *path_leaves* within the execution trees of functions h, g and f.

Function EVAL_NODE makes use of other functions in order to perform symbolic execution. For completeness, these function's algorithms are given immediately following the algorithm of EVAL_NODE.

Note that in the following algorithm, in order to express that an argument is of no consequence when performing a function call, we will be using a generic argument *n_a* which stands for "not applicable". Function EVAL_NODE can be defined as follows (for further explanations, consult the comments given directly into the algorithms):

Function **EVAL_NODE**

(in n, o:V_{CFG};*obj_from* : $VarVal$;vs_g, vs_l : $State$;VAR *path_leaves* : $2^{V_{ET}}$;VAR *computed_val* : *Expression*): *Expression*;
begin
  {*in case of an assignation statement, e.g. "lhs = rhs ;"*}
  if($NODE\_TYPE(n) = Assign$) then begin
    {*modify the value of lhs to be the computation of rhs*}
    {*consider the conditions potentially encountered while computing lhs and rhs*}
    cond_lhs := $TRUE$
    cond_rhs := $TRUE$
    {*evaluate the symbolical memory location of the lhs*}
    lhs := GET_ADDRESS(LHS(n),*obj_from*, vs_g, vs_l, *path_leaves*, *cond_lhs*)
    {*evaluate the current value of the rhs*}
    rhs := GET_VALUE(RHS(n),*obj_from*, vs_g, vs_l, *path_leaves*, *cond_lhs*)
    SET_VAL(*lhs*, *rhs*)

```
    end
    compute the path to be followed in the execution tree of the called function
    V = {v | v ∈ path(FIRST(path_leaves))}
    E = {e =< v1, v2 > | v1, v2 ∈ V ∧ < stm(v1), stm(v2) >∈ E_CFG}
    path_leaves = path_leaves − FIRST(path_leaves)
    {execute the called function/method}
    ec := PBCC_ET(V, E, ROOT(V, E), obj, vs_g, vs'_l, computed_val)
    return(ec ∧ cond_call)
end

if(NODE_TYPE(n) = Return) then begin
    {consider the conditions potentially encountered while computing the returned-value}
    cond_val = TRUE
    computed_val := GET_VALUE(RET_VAL(n),obj_from, vs_g, vs_l, path_leaves, cond_val)
    return(cond_val)
end

if(NODE_TYPE(n) = VarVal) then begin
    {generally, return the value part of the VarVal triplet}
    if (VAL(n) is undefined) then begin
        {return the node n itself, will be encountered after dynamic allocation only}
        computed_val := n
        return(TRUE)
    end else begin
        computed_val := VAL(n)
        return(TRUE)
    end
end

if(NODE_TYPE(n) ∈ {Const, Symbol}) then begin
        computed_val := n
        return(TRUE)
end

if(NODE_TYPE(n) ∈ {Start, Stop}) then begin
        computed_val := n_a
        return(TRUE)
end

if(NODE_TYPE(n) = Unary_exp) then begin
    {consider the conditions potentially encountered while computing the value}
    cond := TRUE
    if (OP(n) = *) then begin
        {obtain the address (the VarVal) corresponding to the child of n}
        computed_val := GET_VALUE(CHILD(n),obj_from, vs_g, vs_l, path_leaves, cond)
    end
    else if (OP(n) = &) then
        computed_val := CHILD(n)
    else
        computed_val := n
    return(cond)
end

if(NODE_TYPE(n) = Binary_exp) then begin
    {ec will accumulate all predicates encountered while computing node n}
```

```
ec = TRUE
if (OP(n) = .) then begin
   {compute the address of LHS(n).RHS(n), to be re-evaluated as needed to obtain the current value}
   base := GET_ADDRESS(LHS(n),obj_from, vs_g, vs_l, path_leaves, cond)
   ec := ec ∧ cond
   computed_val := GET_ADDRESS(RHS(n),base, vs_g, vs_l, path_leaves, cond)
   ec := ec ∧ cond
end
else if (OP(n) = →) then begin
   {compute the address of the field accessed through the pointer (LHS(n)), i.e. the value of LHS(n)}
   base := GET_VALUE(LHS(n),obj_from, vs_g, vs_l, path_leaves, cond)
   ec := ec ∧ cond
   computed_val := GET_ADDRESS(RHS(n),base, vs_g, vs_l, path_leaves, cond)
   ec := ec ∧ cond
end
else if (OP(n) = [ ]) then begin
   {compute the address of the array element LHS(n)[RHS(n)] }
   base := GET_ADDRESS(LHS(n),obj_from, vs_g, vs_l, path_leaves, cond)
   ec := ec ∧ cond
   index_exp := GET_VALUE(rhs,obj_from, vs_g, vs_l, path_leaves, cond)
   ec := ec ∧ cond
   {check for symbolic indexing}
   if (NODE_TYPE(index_exp) = Symbol) then begin
      newindex := EXTERNALLY-OBTAINED-INDEX( ) {reads from a file}
      ec = ec ∧ (index_exp = newindex)
      index_exp := newindex
   end
   computed_val := GET_FIELD(base, index_exp)
end
else if (OP(n) = ::) then begin
   {rhs should be a static attribute or method}
   {compute the address of the static attribute OR the address of the object for which a method is called}
   base := GET_ADDRESS(LHS(n),obj_from, vs_g, vs_l, path_leaves, cond_base)
   ec := ec ∧ cond_base
   computed_val := GET_ADDRESS(RHS(n),base, vs_g, vs_l, path_leaves, cond_obj)
   ec := ec ∧ cond_obj
end else begin
   {Binary expression other than . → [ ] ::}
   lhs := GET_VALUE(LHS(n),obj_from, vs_g, vs_l, path_leaves, cond_lhs)
   rhs := GET_VALUE(RHS(n),obj_from, vs_g, vs_l, path_leaves, cond_rhs)
   ec := ec ∧ cond_lhs ∧ cond_rhs
   {check for pointer arithmetic, e.g. if one side evaluates to a memory location}
   {while the other side is a constant, with operators + and -}
   if (NODE_TYPE(lhs) = VarVal and NODE_TYPE(rhs) = Const and OP(n) ∈ {+, −} ) then begin
      newptr = INC(lhs,rhs)
      {create temporary VarVal that will be reevaluated later, yielding newptr}
      computed_val = < n_a, newptr, n_a >
   end
   {symmetrical case}
   if (NODE_TYPE(rhs) = VarVal and NODE_TYPE(lhs) = Const and OP(n) ∈ {+, −} ) then begin
      newptr = INC(rhs,lhs)
      {create temporary VarVal that will be reevaluated later, yielding newptr}
      computed_val = < n_a, newptr, n_a >
   end
end
```

```
    return(ec)
  end

  if(NODE_TYPE(n) = Identifier) then begin
    {analyze quadruple forming n}
    {note that functions IS-CLASS/STRUCT-ATTRIBUTE, IS-GLOBAL-VARIABLE,}
    {IS-LOCAL-VARIABLE, IS-FUNCTION, IS-METHOD and IS-CLASS are not given }
    {but they basically map n into one of the categories described in 3.2.2}
    if (IS-CLASS/STRUCT-ATTRIBUTE(n)) then
      {compute the VarVal corresponding to that field}
      computed_val := GET_FIELD(obj_from,VAR_ID(n))
    else if (IS-GLOBAL-VARIABLE(n)) then begin
      {compute the VarVal associated to that variable, or create it if inexistent}
      computed_val := vv ∈ vs_g | VAR_ID(VAR(vv))=VAR_ID(n)
      if (computed_val is inexistent) then begin
        computed_val := < n, n_a, n_a >
        vs_g := vs_g ∪ computed_val
      end
    end else if (IS-LOCAL-VARIABLE(n)) then begin
      {compute the VarVal associated to that variable, or create it if inexistent}
      computed_val := vv ∈ vs_l | VAR_ID(VAR(vv))=VAR_ID(n)
      if (computed_val is inexistent) then begin
        computed_val := < n, n_a, n_a >
        vs_l := vs_l ∪ computed_val
      end
    end else if (IS-FUNCTION(n)) then
      {no computation needed}
      computed_val := n_a
    else if (IS-METHOD(n)) then begin
      {return directly the obj_from received}
      computed_val := obj_from
    end else if (IS-CLASS(n)) then
      {create a VarVal in global-level with variable id = -1 * class id. This}
      {way, static attributes will be associated as fields to this newly created VarVal}
      computed_val := vv ∈ vs_g | VAR_ID(VAR(vv))=-1 * CLASS_ID(n)
    return(TRUE)
  end

  if(NODE_TYPE(n) = SystemFctCall) then begin
  {some basic C functions have been defined, for example}
    if C-FUNCTION-NAME(n) = malloc then begin
      {create a new VarVal in the heap (global-level)}
      computed_val := < n_a, n_a, n_a >
      vs_g := vs_g ∪ computed_val
    end
    else if C-FUNCTION-NAME(n) = getch then begin
      {create and return a new symbolic value s with unique identification (name)}
      computed_val := s | s ∈ Symbol
    end
    else if C-FUNCTION-NAME(n) = scanf then begin
      {associate new symbolic values for each of the input variables}
      ∀ input variables v begin
        lhs := GET_ADDRESS(v,obj_from, vs_g, vs_l, path_leaves, n_a)
        SET_VAL(lhs, s) | s ∈ Symbols with s unique
      end
```

```
            computed_val := n_a
        end
        return(TRUE)
    end
end function
```

```
Function INC(in e : Expression; offset : N):Expression;
begin
    {INC increments not only by a value of 1, but by a value of offset (positive or negative)}
    if (NODE_TYPE(e) = VarVal) then begin
        {if e is a pointer, compute element at offset distance from e}
        res := e' | e, e' ∈ fields ∧ VAR_ID(VAR(e')) = VAR_ID(VAR(e)) + offset
    else
        {arithmetic incrementation}
        res := e + offset
    return(res)
end function
```

```
Function SET_VAL(in VAR vv : VarVal; new_val : Expression)
begin
    {modify the value section in the triplet received}
    id' := VAR(vv)
    fields' := FIELDS(vv)
    vv := < id', new_val, fields' >
end function
```

```
Function GET_FIELD(in vv : VarVal; index : Const):VarVal
begin
    {computes and returns field index within VarVal vv or create it if inexistent}
    res := vv' ∈ FIELDS(vv) | index = VAR_ID(VAR(vv'))
    if (res is inexistent) then begin
        res := < n, n_a, n_a >
        FIELDS(vv) := FIELDS(vv) ∪ res
    end
    return(res)
end function
```

```
Function ROOT(in V : 2^V_ET; E : 2^E_ET):V_ET
begin
    {computes and returns the root node of a given tree T = (V, E)}
    v := w ∈ V | ∄ < w', w >∈ E
    return(v)
end function
```

Function **FIRST**(in $< v_1, v_2, ..., v_n >| v_i \in V_{ET}$):$V_{ET}$
begin
  {*returns the first element of a received list*}
  return(v_1)
end function

Function **GET_ADDRESS**
(in $n : V_{CFG}; obj\_from : VarVal; vs_g, vs_l : State; VAR\ path\_leaves : 2^{VET}; VAR\ cond : Expression$):$Expression$
begin
  {*unique evaluation of n, which will give a reference*}
  {*to the VarVal associated to n in memory*}
  cond := EVAL_NODE($n, n\_a, obj\_from, vs_g, vs_l, path\_leaves, address$)
  return(*address*)
end function

Function **GET_VALUE**
(in $n : V_{CFG}; obj\_from : VarVal; vs_g, vs_l : State; VAR\ path\_leaves : 2^{VET}; VAR\ cond : Expression$):$Expression$
begin
  {*evaluation of the address (the VarVal) of n, which will*}
  {*result in the current value of that VarVal*}
  address = GET_ADDRESS($n, obj\_from, vs_g, vs_l, path\_leaves, cond$)
  cond := *cond* $\wedge$ EVAL_NODE($address, n\_a, obj\_from, vs_g, vs_l, path\_leaves, value$)
  return(*value*)
end function

Annexe C

"Advantages of Unconstrained Edges Analysis to Automatic Test Data Generation"

Article écrit par S. Lapierre, E. Merlo, C. Meunier, G. Savard et G. Antoniol et soumis pour publication.

Advantages of Unconstrained Edges Analysis to Automatic Test Data Generation

S. Lapierre[1], E. Merlo[1], C. Meunier[1], G. Savard[1], G. Antoniol[2]

[1] *GEGI, École Polytechnique*, C.P. 6079, Succ. Centre Ville. Montréal, Québec, Canada.
lapierre@casi.polymtl.ca, merlo@rgl.polymtl.ca, meunierc@casi.polymtl.ca, gilles@crt.umontreal.ca

[2] *IRST-Istituto per la Ricerca Scientifica e Technologica*, I-38050 Povo (Trento), Italy.
antoniol@irst.itc.it

ABSTRACT

Among the different approaches to structural software testing, one approach called branch coverage aims at insuring that every branch in the program has been executed at least once. If we are to try to exercise every branch in the program, it would be beneficial to try to minimize the number of paths that would still ensure complete coverage of the program. One such technique consists of finding a minimal set of branches called the set of unconstrained edges.

The technique of finding unconstrained edges is based on the concept of implication between edges in the program; an edge is said to be unconstrained if it does not imply any other edge nor it is implied by any other.

In this article, we evaluate the advantages of unconstrained edges computation in the context of automatic test data generation. In particular, we try to evaluate the difference in complexity between generating test data for unconstrained edges using an automatic generator and generating test data for all program edges. Impacts of infeasible paths are also studied in context of test data generation for unconstrained edges.

We discuss of the algorithms used in our implementation to generate input test data, and demonstrate the basis of the algorithms used. We also show preliminary experimental data to confirm the reduction in the efforts required for such an automatic generation based on unconstrained edges computation. Such experimental data and discussion represent an originality, being analyzed in the precise context of a development and testing environment. Some of these effort measures are based on the number of paths and number of statements executed by paths.

1 INTRODUCTION

The goal of software testing is to detect bugs in a program under test, in order to remove defects and increase software quality. This activity consumes at least half of the labor expended to produce a working program [3], thus requiring an enormous amount of resources of all kind (human resources, time and money) for this quality to be achieved.

Two major testing methods are available during the testing phase and are termed functional testing (black box testing) and structural testing (white box testing).

In functional testing, only the functionality of a software component is tested; during this process, the tested component is executed with given input data and the produced output is verified for conformity according to an oracle (the program's specifications for example), which specifies the expected outcome of a set of tests. Functional testing thus analyzes a software component with the user's point of view, being interested in what the program does rather than in how it does it. This second concern is the focus point for structural testing, which relies on the internal structure of the tested software component, e.g. on it's implementation details, in order to proceed with testing. Structural testing aims at producing input data which will enable the coverage of a given testing criterion, data that will afterwards be used as inputs during a functional testing phase. Ultimately, structural testing should produce input data that will ensure execution of all program paths. This coverage criterion is impossible to attain, due to loop termination problems [3] and infeasible paths. Structural testing criteria are therefore softened, given goals such as program statement coverage, program branch coverage, or data-flow criteria such as definition-use coverage [12].

The goal of branch testing is to exercise every branch alternative in the program at least once. An input data set permitting to attain such a criterion could contain an input case for each program branch. The size of the input set and time required for their computation would consequently grow as a function of the number of program branches. In [1, 4, 5], techniques are presented

which reduces the number of program branches to be covered while still potentially attaining full branch coverage. These techniques are based on concepts of implication between edges in the program; intuitively, if all possible executions of the program passing through edge e_2 also pass through edge e_1, we can say that an input data case covering edge e_2 also covers edge e_1. Essential branches, called unconstrained edges, are therefore identified within a program, with the particularity that coverage of these branches leads to full branch coverage. Note that the terminology used in this article is taken from [4].

In this article is presented an evaluation of the advantages of unconstrained edges computation in context of automatic test data generation. In particular, using an approach for automatic test data generation based on symbolic execution of paths composing an execution tree (see section 4 for more details), we show preliminary data that suggest a reduction in the efforts required for such an automatic generation due to unconstrained edges computation.

In section 2, the method intuitively presented for unconstrained edge computation is described more formally, followed by the description of it's implementation in section 3. Section 4 presents an overview of our automatic test data generator and explains the comparison criteria that will be used in order to evaluate the contribution of unconstrained edges in automatic test data generation. Experimental results are presented in section 5, discussed in section 6 and a conclusion is given in section 7.

2 DESCRIPTION OF UNCONSTRAINED EDGES APPROACH

This section presents an overview of the concepts involved in unconstrained edges computation [1, 4, 5].

To help the reader better understand the presented concepts, examples will be given at each step of the unconstrained edges generation process. For these examples, program WordCount [6] has been selected. This program is presented in figure 1.

2.1 Program graphs

In order to compute unconstrained edges, different types of program representation are needed.

2.1.1 CFG_nodes A *CFG_nodes* is a control flow graph representing the logical structure of a program and which can be defined as follows:

$$CFG\_nodes = (V_n, E_n)$$

In this structure, each node in V_n is associated with a statement in the program and each edge in E_n connecting nodes of V_n represents a possible flow of control between statements.

```
    void main(void)
    {
      int inword, nl, nw, nc, c ;

 1 :  inword = 0 ;
 2 :  nl = 0 ;
 3 :  nw = 0 ;
 4 :  nc = 0 ;
 5 :  c = getc(stdin) ;
 6 :  while(c != EOF)
      {
 7 :    nc = nc + 1 ;
 8 :    if (c == '\n')
 9 :      nl = nl + 1 ;
10:    if (c=='\n' || c=='\t' || c==' ')
11:      inword = 0 ;
12:    else if (inword == 0) {
13:      inword = 1 ;
14:      nw = nw + 1 ;
       }
15:    c = getc(stdin) ;
      }
16:  printf("value of nl : %d",nl) ;
17:  printf("value of nw : %d",nw) ;
18:  printf("value of nc : %d",nc) ;
    }
```

Figure 1: *Code for WordCount program*

The following holds:

$$(e =< v_1, v_2 > \in E_n) \rightarrow (v_1 \in V_n) \wedge (v_2 \in V_n)$$

CFG_nodes for WordCount program is given in figure 2.

2.1.2 CFG_edges A *CFG_edges* is another representation for a program control flow graph which can be defined as follows:

$$CFG\_edges = (V_e, E_e)$$

In this representation, each node in V_e is associated with a flow of control between statements in the program. In other words, a distinct node v_e exists in V_e for each edge e_n of E_n. Given the following function $Edge : V_e \rightarrow E_n$, which establishes the relation between each node of V_e and it's associated edge in E_n, it can be stated that $\forall e \in E_n,\ \exists v \in V_e \mid Edge(v) = e$.

An edge e_e of E_e connecting nodes v_{e1} and v_{e2} of V_e

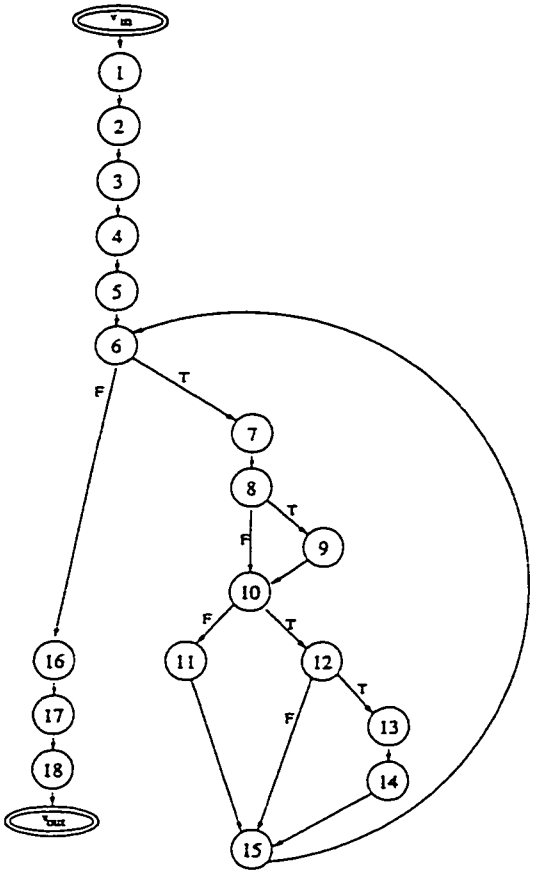


Figure 2: *CFG_nodes for WordCount program.*

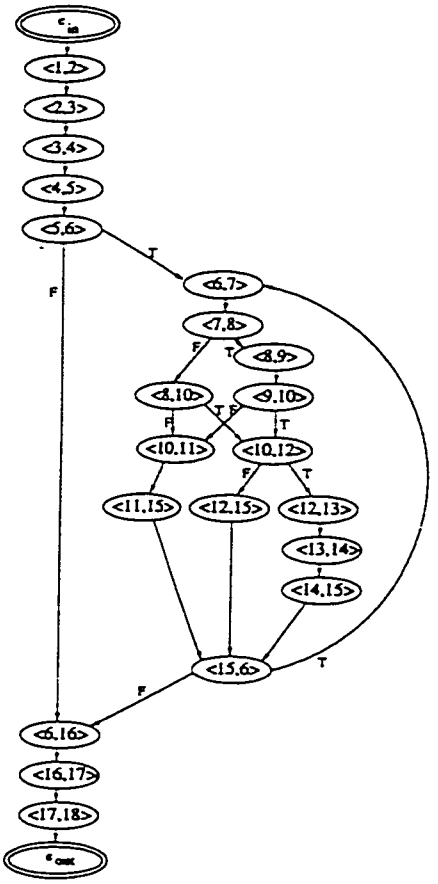


Figure 3: *CFG_edges for WordCount program.*

represents the possibility that control can directly pass from a control flow (v_{e1}) to another (v_{e2}), which corresponds to the following condition:

Let $e_e =< v_{e1}, v_{e2} >$,

$$e_e \in E_e \rightarrow v_{e1} \in V_e \wedge v_{e2} \in V_e \wedge < v_{n1}, v_{n2} >= Edge(v_{e1}) \wedge < v_{n3}, v_{n4} >= Edge(v_{e2}) \wedge v_{n2} = v_{n3} \quad (1)$$

CFG_edges for WordCount program is given in figure 3.

2.1.3 ddgraph A ddgraph is a graph representation that was used in [4] and which corresponds to a *CFG_edges* where each edge represents a program segment. Program segments correspond to basic blocks [2] and are sequences of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching.

Let *CFG_ddgraph* be such a ddgraph and $CFG\_ddgraph = (V_d, E_d)$.

The following holds:

$$(e =< v_{d1}, v_{d2} >\in E_d) \rightarrow (v_{d1} \in V_d) \wedge (v_{d2} \in V_d)$$

and

$$\forall e \in E_d, \ \exists < v_1, v_2, \ldots, v_n > | \ \forall i = 1..(n-1) \ v_i \in V_n \wedge < v_i, v_{i+1} >\in E_n \wedge \nexists < v_i, v_j >\in E_n \mid v_j \neq v_{i+1}$$

2.2 Program path

A path in a graph $G = (V, E)$ can be defined as any possible sequence of nodes of V connected with edges of E, e.g. let p be a path,

$$p =< v_1, v_2, .., v_n > | \forall i = 1..n-1, \exists < v_i, v_{i+1} > \in E \quad (2)$$

2.3 Dominance and postdominance relations

Let $G = (V, E)$ be a graph with unique entry node $v_{in} \in V$ and unique exit node $v_{out} \in V$ such that

$$v_{in} = v \in V \mid \nexists < w, v > \in E$$

and

$$v_{out} = v \in V \mid \nexists < v, w > \in E$$

We can say that:

$$v_1 \; dom \; v_2 \rightarrow \forall p =< v_{in}, .., v_2 >, v_1 \in p \quad (3)$$

and

$$v_1 \; pdom \; v_2 \rightarrow \forall p =< v_2, .., v_{out} >, v_1 \in p \quad (4)$$

for arbitrary nodes $v_1 \in V$ and $v_2 \in V$.

In other words, node $v_1 \in V$ dominates node $v_2 \in V$ if every path in G from v_{in} to v_2 goes through v_1, and node $v_1 \in V$ postdominates node $v_2 \in V$ if every path in G from v_2 to v_{out} goes through v_1.

Dominance and postdominance relations [7, 8, 10, 11] can be refined to express immediate dominance *idom* and immediate postdominance *ipdom*. A node v_1 immediately dominates node v_2 if v_1 dominates v_2 and there is no other dominator of v_2 between v_1 and v_2. More formally,

$$v_1 \; idom \; v_2 \rightarrow \quad v_1 \; dom \; v_2 \wedge \nexists w \in V \mid w \; dom \; v_2 \wedge \exists p =< v_{in}, .., v_1, .., w, .., v_2 > \quad (5)$$

Immediate postdominance is a symmetrical relation to the one just expressed. A node v_1 immediately postdominates node v_2 if v_1 postdominates v_2 and there is no other postdominator of v_2 between v_1 and v_2. More formally,

$$v_1 \; ipdom \; v_2 \rightarrow \quad v_1 \; pdom \; v_2 \wedge \nexists w \in V \mid w \; pdom \; v_2 \wedge \exists p =< v_2, .. w, .., v_1, .., v_{out} >$$

2.4 Dominator and postdominator trees

Given a graph $G = (V, E)$, immediate dominance relations between nodes of V can be expressed in a tree structure, called a dominator tree (*dom_tree*). In this tree, the nodes are arranged in such a way that children nodes immediately dominate their direct parent node. More formally, let $dom\_tree = (V_{dt}, E_{dt})$ be the dominator tree associated to G, $Node : V_{dt} \rightarrow V$ be a function that establishes the correspondence between a node in *dom_tree* and it's associated node in G and $Parent : V_{dt} \rightarrow V_{dt}$ be a function that returns the parent node of a given node in *dom_tree*. We have:

$$V_{dt} = \{v \mid Node(v) \in V \wedge Node(v) \; idom \; Node(Parent(v))\} \quad (6)$$

Postdominance relations can also be expressed in a tree structure similar to *dom_tree*. We will call $pdom\_tree = (V_{pdt}, E_{pdt})$ the postdominator tree computed for a given graph $G = (V, E)$. Given functions $Node : V_{pdt} \rightarrow V$ which establishes the correspondence between a node in *pdom_tree* and it's associated node in G and $Parent : V_{pdt} \rightarrow V_{dt}$ which returns the parent node of a given node in *pdom_tree*, we have:

$$V_{pdt} = \{v \mid Node(v) \in V \wedge Node(v) \; ipdom \; Node(Parent(v))\} \quad (7)$$

2.5 Unconstrained edges

The set of unconstrained edges UE is defined as the set of edges in a graph $G = (V, E)$ that dominate no other edge and that post-dominated no other edge, which corresponds to the following equation:

$$\begin{array}{c} < v_a, v_b > \in UE \Leftrightarrow \\ (\nexists < v_i, v_j > \in E \mid \\ (\forall p =<< v_{in}, v_a >, ..., < v_i, v_j >> \rightarrow < v_a, v_b > \in p)) \wedge \\ (\nexists < v_r, v_s > \in E \mid \\ (\forall q =<< v_r, v_s >, ..., < v_m, v_{out} >> \rightarrow < v_a, v_b > \in q)) \end{array} \quad (8)$$

where $< v_a, v_b > \in E$ is an edge in G, v_{in} and v_{out} are the starting and ending edges in G.

Note that this definition of unconstrained edges is equivalent to the concept of essential branches based on super blocks, as defined in [1]

The set UE therefore represents the minimal set of edges which have to be covered in order to achieve total branch

coverage of the program to be analyzed. See [1, 4, 5] for further details.

Given the previous definitions, the set UE can be computed as the intersection of the leaves of the dominator tree of CFG_edges with the leaves of the postdominator tree of CFG_edges.

Let $CFG\_edges = (V_e, E_e)$, $dom\_tree = (V_d, E_d)$ be the dominator tree computed for CFG_edges and $pdom\_tree = (V_p, E_p)$ be the postdominator tree computed for CFG_edges,

$$UE = \left\{ \begin{array}{l|c} v_d \in V_{dt} & \exists v_p \in V_{pdt}, \\ & Node(v_d) = Node(v_p) \wedge \\ & \nexists w_d \in V_{dt} \mid v_d = Parent(w_d) \wedge \\ & \nexists w_p \in V_{pdt} \mid v_p = Parent(w_p) \end{array} \right\}$$

2.6 Program edge path

An edge path is also defined here. An edge path in $G = (V, E)$ can be defined as any possible sequence of edges of E connected by nodes of V, e.g. let p be a path,

$$\begin{aligned} p \;=\; & < e_1, e_2, \ldots, e_n > \mid \\ & e_i =< v_{2i}, v_{2i+1} > \in E,\ \forall i = 1 \ldots n, \\ & v_{2i}, v_{2i+1} \in V, \\ & v_{2j+1} = v_{2(j+1)},\ \forall j = 1 \ldots n-1 \end{aligned} \tag{9}$$

2.7 Edge dominance

We will also define the dominance and postdominance relations on the edges of a graph. We will consider a graph $G = (V, E)$ with unique entry edge $e_{in} \in E$, and a unique exit edge $e_{out} \in E$:

$$\begin{aligned} e_{in} \;=\; & e \in E \mid \\ & e =< v_{in}, v >, v_{in} \in V, \\ & \nexists e_2 \in E, e_2 =< w, v_{in} > \end{aligned}$$

and

$$\begin{aligned} e_{out} \;=\; & e \in E \mid \\ & e =< v, v_{out} >, v_{out} \in V, \\ & \nexists e_2 \in E, e_2 =< v_{out}, w > \end{aligned}$$

That is, e_{in} is the unique edge coming out of v_{in}, and e_{out} is the unique edge going to v_{out}. If no unique edge exist, additional nodes are created before v_{in} or after v_{out} and edges are added, to create unique entry end exit edges.

To define dominator and postdominance relations on the edges of G, we can say:

$$e_1 \; dom \; e_2 \leftrightarrow \forall p =< e_{in}, \ldots, e_2 >, e_1 \in p \tag{10}$$

and

$$e_1 \; pdom \; e_2 \leftrightarrow \forall p =< e_2, \ldots, e_{out} >, e_1 \in p \tag{11}$$

That is, if e_1 dominates e_2, then e_1 is on every edge path going from e_{in} to e_2; if e_1 postdominates e_2, then e_1 is on every edge path from e_2 to e_{out}.

2.8 Edge graph construction

Let us show that when we calculate the dominator and postdominator tree on the CFG_edge we defined, we obtain the dominator and postdominator relations defined in section 2.7.

Let P be a program to be analyzed, $G_n = (V_n, E_n)$ be it's CFG_nodes and $G_e = (V_e, E_e)$ be it's CFG_edges. We must show that when we apply the dominance/postdominance algorithms to the edge graph G_e, we obtain the dominance/postdominance relations on the edges of the CFG_nodes G_n. In other words, we must obtain a dominator tree in which the nodes of the tree will give us the edge-dominance relations in P.

2.8.1 Dominance Let's demonstrate that an edge $e_1 \in E_n$ dominates another edge $e_2 \in E_n$ if and only if, in the edge graph, the node $v_{e1} \in V_e$ dominates the node $v_{e2} \in V_e$. That is,

$$\begin{aligned} e_1 \; dom \; e_2 \;\Leftrightarrow\; & v_{e1} \; dom \; v_{e2}, \\ & e_1 = Edge(v_{e1}), e_2 = Edge(v_{e2}) \end{aligned}$$

If e_1 *dom* e_2 in G_n: Let us prove that e_1 *dom* e_2 in G_n implies that v_{e1} *dom* v_{e2} in G_e.

If e_1 *dom* e_2 then, by equation 10, e_1 will be on all edge paths in G_n from e_{in} to e_2:

$$\forall p, p =< e_{in}, \ldots, e_2 >, e_1 \in p$$

Let's translate any one of these paths p into an ordered list l, keeping the same ordering as in p. We obtain:

$$l =< v_{ein}, \ldots, v_{e2} >$$

where $e_i = Edge(v_{ei}), \forall e_i \in l$. Note that v_{e1} will always be in l, since $e_1 \in p$.

Since p is an edge path in G_n, then by equation 9, there exists a node $v \in V_n$ between each consecutive pair of edges in p.

Since we have a node in V_n between any two consecutive edges, then by equation 1 there will be an edge $e_{ei} \in E_e$ between each of the nodes of l.

The list l will then be a path in G_e. Since l is any path in G_e from v_{e1n} to v_{e2}, and $e_1 \in l$, we can say that v_{e1} *dom* v_{e2} in G_e. We have proven the first part of the implication.

If v_{e1} *dom* v_{e2} in G_e: Let us prove that v_{e1} *dom* v_{e2} in G_e implies that e_1 *dom* e_2 in G_n.

If v_{e1} *dom* v_{e2} then, by equation 3, v_{e1} will be on all paths in G_e from v_{e1n} to v_{e2}:

$$\forall p, p =< v_{e1n}, \ldots, v_{e2} >, v_{e1} \in p$$

Take any of these paths p. Then take the ordered list $l =< e_{in}, \ldots, e_2 >$ of edges of E_n, where $e_i = Edge(v_{ei}), \forall v_{ei} \in l$. Note that if $v_{e1} \in p$, then $e_1 \in l$.

Since p is a path in G_e, by equation 2 there exists an edge e between each nodes of p. Take any two of these nodes, v_1 and v_2; $e_1 =< v_1, v_2 >$.

Since this edge exists in E_e, then by equation 1 there exists two edges $e_{n1} =< n_1, n_2 >$ and $e_{n2} =< n_3, n_4 >$ in E_n such that $n_2 = n_3$, with $n_1, n_2, n_3, n_4 \in V_n$.

From the same reasoning, l will be an edge path in G_n. Since any edge path from e_{in} to e_2 in G_n has $e_1 \in l$, then e_1 *dom* e_2 in G_n.

We have thus proven the correspondence between edge dominance relation on a graph and the dominance relation applied on a corresponding edge graph.

2.8.2 Postdominance
To demonstrate that e_1 *pdom* e_2 in G_n if and only if v_{e1} *pdom* v_{e2} in G_e, the proof is fairly similar to the one for the dominance relation. It is left as an exercise to the reader.

2.9 Using edge flow graphs

In [4], it is shown that the set of unconstrained edges can be obtained by taking the intersection UE of the set of leaves from the dominator tree and of the postdominator tree, both taken on the graph $CFG\_ddgraph$. We show here the particularities that arise from using node graphs.

Bertolino uses the function *reduce* to convert an arbitrary digraph $G = (V, E)$ into a ddgraph $CFG\_ddgraph$. The effect of this algorithm is as follows: it contracts into a single node every node path in G consisting of:

- one node having exactly one outgoing edge, and no or many incoming edges;
- any number of nodes having exactly one incoming and one outgoing edge;
- one node having at most one incoming edge, and no or many outgoing edges.

Formally, a path p in G is contracted into a single node n' of $CFG\_ddgraph$ if it is the longest path such that:

$$\begin{aligned} p \;=\; & < v_1, \ldots, v_n >, \qquad (12) \\ & \exists! e_1 =< v_1, v >\in E, e_n =< w, v_n >\in E \wedge \\ & \exists! e_i =< v_i, x >\in E \wedge \exists! e_i =< y, v_i >\in E, \\ & \quad i = 2 \ldots n-1 \end{aligned}$$

Let us define the functions $In : V \rightarrow \mathcal{N}$ and $Out : V \rightarrow \mathcal{N}$. The function In returns the number of edges of G coming into a node v, and the function Out returns the number of edges coming out of a node v:

$$In(v) = card(\{e =< u, v >| \; e \in E\})$$

and

$$Out(v) = card(\{e =< v, u >| \; e \in E\})$$

We wish to determine, without using **reduce**, the same set of nodes that is generated by the following operations:

1. calculate $CFG\_ddgraph = reduce(G)$
2. $UE \;=\; leaves(dom(CFG\_ddgraph)) \cap leaves(pdom(CFG\_ddgraph))$

Let us call p one of the basic blocks, $p =< v_1, \ldots, v_n >$. Let us look at how it is represented into *dom_tree* and *pdom_tree*.

2.9.1 Case where $|\; p \;| = 1$ For the proof, we will not elaborate on the trivial case where an element of UE does not come from the contraction of a node as a product of the function **reduce**. Such an element is assuringly a node of *dom_tree* and *pdom_tree*, thus an element of UE.

2.9.2 Representation of p in dom_tree Let us take the first node of p, v_1. By equation 12,

$$In(v_1) \neq 1, Out(v_1) = 1.$$

Since it has only one child v_2 (i.e. $v_1 = Parent(v_2)$), then node v_1 will be on every path to v_2. We can then

say that v_1 *dom* v_2. Moreover, by equation 5, v_1 *idom* v_2 since no other node will be on any path between v_1 and v_2.

We then take the next nodes of p, the nodes $v_2, \ldots, v_{n-1}$. The following holds true for all these nodes:

$$In(v) = Out(v) = 1.$$

Thus for $i = 2 \ldots n-1$, each of these nodes v_i has only one parent, the node v_{i-1}.

Every path to node v_i will then include v_{i-1} as its penultimate node; we can thus say that

$$v_{i-1} \; idom \; v_i, i = 2 \ldots n-1.$$

Take then the last node v_n of p, $In(v_n) = 1, Out(v_n) \neq 1$. Since it has only one parent v_{n-1}, then v_{n-1} *idom* v_n.

Thus, for every node of p, we have:

$$v_{i-1} \; idom \; v_i, i = 2 \ldots n.$$

By equation 6, the path p will be expressed as a path p_2 in the same order as is the path p, with:

$$p_2 =< v'_1, \ldots, v'_n >$$

where

$$p =< Node(v'_1), \ldots, Node(v'_n) >$$

2.9.3 Representation of p in pdom_tree The same reasoning applies with the representation of p in *pdom_tree*. Since $Out(v) = 1$ for all nodes $v = v_1, \ldots, v_{n-1}$, then these nodes will immediately postdominate their respective child, that is:

$$v_{i+1} \; ipdom \; v_i, i = 1 \ldots n-1.$$

Thus by equation 7, the path p of G is expressed as a path p_2 in *pdom_tree*:

$$p_2 =< v'_n, \ldots, v'_1 >$$

where

$$p =< Node(v'_1), \ldots, Node(v'_n) >$$

We have proven that if we use directly an edge graph instead of a ddgraph to calculate *dom_tree* and *pdom_tree*, we will find the nodes representing a basic block in non-branching paths of the trees.

This then guides us to the following way to find the intersection of nodes UE: find a maximal non-branching path that leads to a leaf of *dom_tree*, and then look for this path, inverted, in *pdom_tree*. If the inverted path leads to a leaf of *pdom_tree*, then the basic block this path represents is in UE. Take any node of the path to represent all the nodes in that basic block. This is the basis for the algorithm ue, presented in the next section.

3 UNCONSTRAINED EDGES COMPUTATION

In this section, we will describe the tools that have been developed to run the experiments on the unconstrained edges.

3.1 Overview

In Figure 4, we can see a global view of the tools that were used to generate the list of unconstrained edges. Each of these tools is described briefly below.

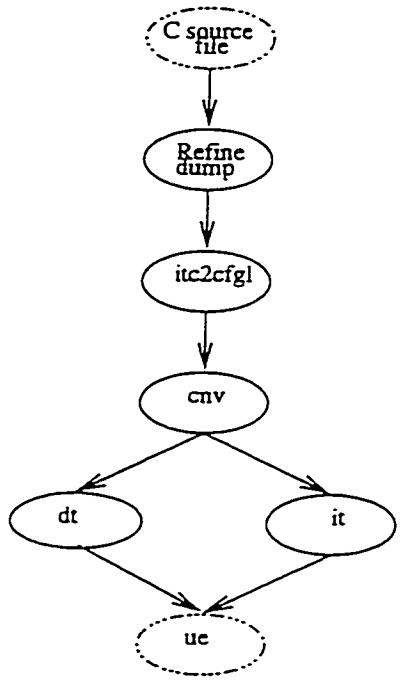


Figure 4: *Data flow through the tools.*

3.2 Parsing

The first stage in the generation of a list of unconstrained edges is to parse the program under analysis. For this task, a parse tree is built using Refine [1], and then saved on disk for further processing.

[1] Refine is a trademark of Reasoning Systems Inc.

3.3 Creating the node graph

This second stage, which uses a tool we called *itc2cfgl*, reads back the parse tree generated from Refine and translates it to a *CFG_node*. This tool has been built around a yacc parser, and reads its input from a parse tree cleaned-up by a Perl script.

3.4 Converting to an edge graph

This third stage in the process generates a *CFG_edge* from the *CFG_node* generated in the previous phase. The algorithm used to do this task extracts every possible flow control from one edge to another in the *CFG_node*.

The algorithm is given in figure 5.

```
procedure Conversion( Gin: Graph, Gout:Graph)
  // Gather connections between nodes
  for each edge e=<v1,v2> in Gin
    add edge e to the lists Hv1 and Tv2

  // Produce the new graph
  for each node n of Gin
    for each element v1 in Hn
      for each element v2 in Tn
        add edge <v1,v2> in Gout
      end for
    end for
  end for
end procedure
```

Figure 5: *Converting the node graph to an edge graph.*

3.5 Dominator trees

The immediate dominator and postdominator trees are then calculated on the graph generated by the conversion.

3.6 Unconstrained edges

The last stage calculated the intersection of the leaves of the dominator and postdominator trees. Some leaves are considered equivalent for the purpose of this intersection; we refer to the concept of extended leaf described earlier.

The algorithm used here tries to determine if a given node a, taken from the set of the leaves of the immediate dominator tree, is an extended leaf in the postdominator tree. It does so by finding the associated node in the postdominator tree, then determine if this node is on a straight path to a leaf of the postdominator tree.

The algorithm is reproduced in figure 6.

```
// Determine the set of unconstrained edges
// from dom_tree and pdom_tree
function UnconstrainedEdges(dom_tree, pdom_tree
  A = leaves(dom_tree)
  UE = empty set

  for each a in A
    if isExtendedLeaf(a, pdom_tree)
      UE = UE union a
    end if
  end for

  UnconstrainedEdges = UE
end function

// determine if node 'a' is on a straight path
// to a leaf of pdom_tree
function isExtendedLeaf(a, pdom_tree)
  x = the node in pdom_tree which
      corresponds to 'a'
  done = FALSE
  isExtendedLeaf = TRUE

  while !done
    if x has only one child
      x = x's child
    else if x has no child
      done = TRUE
    else // x has more than one child
      isExtendedLeaf = FALSE
      done = TRUE
    end if
  end while
end function
```

Figure 6: *Finding Unconstrained Edges.*

4 USE OF UNCONSTRAINED EDGES IN TEST CASE GENERATION

As stated in section 1, unconstrained edges information can be used in testing in order to reduce testing efforts while still potentially achieving full branch coverage. In this study, we are interested to evaluate such impacts of unconstrained edges on automatic test case generation. In particular, we will try to evaluate the reduction in effort required to achieve test data generation while using unconstrained edges information.

We will be using an automatic test data generator (*TAO*) based on symbolic execution of paths in an execution tree, which will be briefly described in the following subsection.

4.1 Automatic test data generator

Our automatic test data generator, *TAO*, performs an inter-procedural search in a program graph (CFG_nodes) in order to find paths which lead to a given edge e and which have limited and controlled iteration limits. The collection of paths thus computed to reach edge e can be united in a tree representation called a partial execution tree. A characteristic of this tree is that it's leaves correspond to goal edge e.

Symbolic execution is then performed on each path included in this tree. Symbolic execution can be described as a simulation of an execution, along a given program path, where each statement's actions are simulated. A special characteristic of symbolic execution is that input statements introduce unknown data values, which are left unknown and are termed as "symbols". Conditional statements that are encountered along the path are evaluated and their predicates are accumulated into a *pc* (path condition) which is a logical expression, function of input symbols, which has to be validated for the path to be traversed. In our system, symbolic execution of the paths which compose a partial execution tree yields a tree decorated with the predicate conditions encountered while performing symbolic execution. This decorated tree is thus composed of different symbolically conditioned paths; the goal of test data generation is to find input data to force execution along any one of these paths. This corresponds to finding values for symbols which will validate any one of the *pcs* of the paths in the execution tree. The *pcs* are transformed into a logical symbolic expression tree with the particularity that a branching in the tree introduces a disjunction in the expression "$\vee$" (one path OR the other is to be validated) while non-branching nodes introduce a conjunction "$\wedge$".

The final stage of the presented process consists in solving the logical symbolic expression. To do so, we transform the logical problem into a linear optimization problem and make use of an optimization tool which tries to find values for symbols which optimize the equation (in our case, the optimization factor is irrelevant and it's validation is sufficient).

A detailed description of *TAO* test data generation system can be found in [9].

4.2 Criteria for evaluation of unconstrained edges impact on test case generation

In order to evaluate the impact of unconstrained edges in automatic test data generation, we will compare automatic generation of test data using *TAO* for unconstrained edges and for all edges on the following terms:

- reduction in the number of goal edges to be covered (ratio of unconstrained edges on all edges)
- comparison of the size of the average partial execution trees generated for unconstrained edges and for all edges, in terms of number of possible paths and number of statements in the tree.
- comparison of the size of the logical problem generated for unconstrained edges and for all edges, in terms of number of symbols implied.
- time required to generate a test suite achieving coverage of the desired goal (all edges or all unconstrained edges)

5 EXPERIMENTAL RESULTS

A total of 12 C-language programs have been analyzed. These programs have been selected because they contain special characteristics that are somewhat problematic in automated test data generation. Such characteristics include the treatment of symbolic indexing and the use of pointers, function pointers and dynamic memory operations [9].

Table 1 gives some characteristics for these programs, namely their name and their size in LOC (number of lines of code) measured as the number of *end-of-line* characters in the program source files.

Program	LOC
1. wc	39
2. patternmatcher	77
3. prog6	68
4. taotest	239
5. minpath	194
6. quicksort	95
7. histogramme	45
8. what	130
9. trityp	62
10. exptree	389
11. nombre	67
12. inves	57
total	1462

Table 1: *Characteristics of the programs analyzed.*

The programs were analyzed in order to determine their unconstrained edges. Table 2 gives the number of edges for each program (which corresponds to the number of edges in the program CFG_{nodes}), the number of unconstrained edges computed for each program, and the ratio between these two measures(for each program), which corresponds to:

$$ratio = \frac{number\ of\ unconstrained\ edges}{number\ of\ edges}$$

In context of software branch testing, results of table 2 show us that unconstrained edges computation enables us to reduce the number of edges to be reached for potentially attaining 100% branch coverage by a factor of 80% on the tested sample.

program edges. Results indicate no major differences between the two set of analyzed edges, whether when considering the average number of paths, the average number of statements or the average number of symbols implied in the partial execution trees generated for coverage of a given edge. Arising from these results is the conclusion that, on average, generating test data for an unconstrained edge is approximately on the same level of difficulty as generating test data for any one of the program edges.

This last result is of interest when considering that unconstrained edges represent 20%-29% of all program edges; if test data generation is, on average, of the same difficulty for edges in the unconstrained edges set or for all program edges, then obviously test data generation should aim at covering only the edges in the set of unconstrained edges, e.g. 20%-29% of all program edges, with no added difficulty and with roughly a 77% reduction in time needed to generate data for high program coverage.

Infeasible paths in programs are certainly a major problem while generating test data. Primarily, the difficulty lies in the detection of an infeasible path, which is an undecidable problem. Nevertheless, an analysis of the programs in our testbed shows that selecting only a program's unconstrained edges does not increase the ratio:

$$\frac{\textit{number of edges to be treated} \in \textit{some infeasible path}}{\textit{total number of edges to be treated}}$$

This result clearly indicates that no added difficulty related to infeasible paths arises when dealing solely with unconstrained edges.

Given the preliminary results obtained from this experiment, any test data generation effort, whether done automatically or by hand, should aim at covering only the edges of the unconstrained edges set. Furthermore, no added difficulty should rise from analyzing only such unconstrained edges. On the contrary, the only impact noticeable while generating the test data should be an important reduction in time to achieve a high degree of branch coverage.

7 CONCLUSION AND FURTHER RESEARCH

In this article, we have shown preliminary experimental results on the advantages of unconstrained edges in the automatic generation of input test data for branch coverage. Using the presented technique, which relies on dominance/postdominance relations in a graph, thus represents a very effective way to reduce general test data generation efforts.

Based on these results, it seems no more difficult to generate the data for paths containing unconstrained edges than it is to generate it for all the paths of the program; with the added benefit that a much smaller set of paths has to be generated to obtain a full branch coverage of the program. Branch coverage based on the set of unconstrained edges in the program then should be a good way to significantly reduce the number of paths to be executed during the coverage tests.

Further research efforts include a more extensive experimental validation of the preliminary conclusions drawn from the present results.

REFERENCES

[1] H. Agrawal. "Dominators, Super Blocks, and Program Coverage", *POPL'94:Symp. on Principles of Programming Languages*, 1994, pp.25–34.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. "Compilers: Principles, Techniques, and Tools", Addison-Wesley, 1986.

[3] B. Beizer. "Software testing techniques, second edition", International Thomson computer press, 1990.

[4] A. Bertolino and M. Marré. "Automatic generation of path covers based on the control flow analysis of computer programs", *IEEE Trans. Soft. Eng.*, dec. 1994, pp.885–898.

[5] T. Chusho. "Test data selection and quality estimation based on the concept of essential branches for path testing", *IEEE Trans. Soft. Eng.*, may 1987, pp.509–517.

[6] K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance", *IEEE Trans. Soft. Eng.*, 17(8), 1991, pp.751–761.

[7] R. Gupta, "Generalized dominators and postdominators", *Conference record of the nineteenth ACM symp. on principles or prog. lang. (POPL'92)*, jan. 1992, pp.246–257.

[8] D. Harel, "A linear time algorithm for finding dominators in flow graphs and related problems", *Proc. 17th ACM symp. on theory of computing*, may 1985, pp.185–194.

[9] S. Lapierre, E. Merlo, G. Savard, G. Antoniol, R. Fiutem and P. Tonella. "Description of *TAO* test data generation system", *École Polytechnique technical report*,EPM/RT-97/25, aug. 1997.

[10] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph", *ACM trans. on programming lang. and syst.*, july 1979, pp.121–141.

[11] P. W. Purdom Jr. and E. F. Moore, "Immediate predominators in directed graphs", *Comm. of the ACM*, aug. 1972, pp.777-778.

[12] S. Rapps and E. Weyuker. "Data flow analysis techniques for test data selection", *IEEE proc. of ICSE-6*, 1982, pp.272-278.

Program	Average number of paths in partial execution tree	
	for all Edges	for edges in UE
wc	5.21	1.53
patternmatcher	2.29	3.00
prog6	4.30	5.00
taotest	2.87	2.65
minpath	13.05	16.31
quicksort	2.22	2.88
histogramme	2.62	3.33
what	3.91	4.20
tritvp	52.42	18.74
exptree	8.67	13.13
nombre	10.95	13.33
inves	1.25	1.00
average	9.15	7.09

Table 3: *Average number of paths in partial execution tree for all analyzed programs.*

Program	Average number of statements in partial execution tree	
	for all Edges	for edges in UE
wc	41.18	7.60
patternmatcher	20.00	23.50
prog6	36.59	39.75
taotest	35.20	21.06
minpath	95.77	217.90
quicksort	27.77	50.12
histogramme	47.95	40.33
what	33.80	37.00
tritvp	344.66	173.90
exptree	90.80	294.47
nombre	63.47	65.33
inves	7.43	4.00
average	70.39	81.25

Table 4: *Average number of statements in partial execution tree for all analyzed programs.*

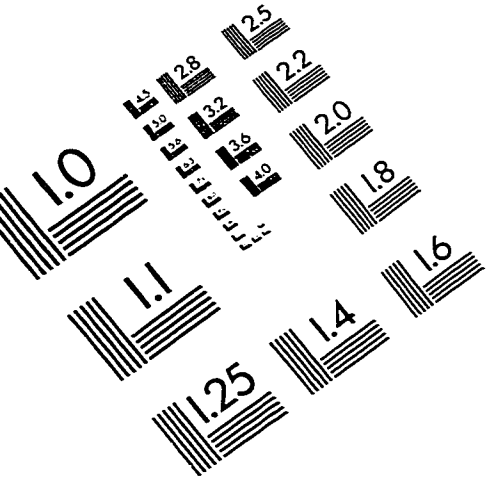
Program	Average number of symbols implied in partial execution tree	
	for all Edges	for edges in UE
wc	1.53	1.80
patternmatcher	2.56	3.14
prog6	1.59	2.00
taotest	3.11	3.53
minpath	3.26	4.47
quicksort	2.88	3.38
histogramme	1.11	1.33
what	6.07	6.80
tritvp	3.00	3.00
exptree	2.08	2.37
nombre	2.06	2.00
inves	0.00	0.00
average	2.44	2.82

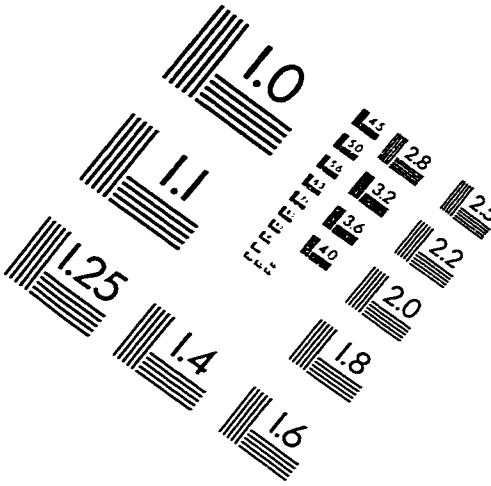
Table 5: *Average number of symbols implied in partial execution tree for all analyzed programs.*

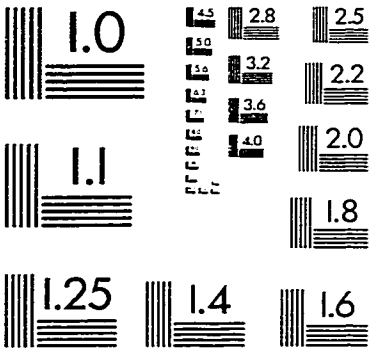
Program	coverage achieved for edges ∈ UE	Time required (sec.)	coverage achieved for all edges	Time required (sec.)
wc	100	1.15	100	3.7
patternmatcher	100	1.48	100	7.2
prog6	100	0.68	100	5.1
taotest	100	11.23	100	62.0
minpath	94	14.97	98	88.5
quicksort	100	2.70	100	16.6
histogramme	100	0.78	100	6.2
what	100	1.29	100	8.7
trityp	94.7	12.67	97	66.8
exptree	100	36.5	100	179.6
nombre	100	0.63	100	4.3
inves	100	0.07	100	0.9
average	99.06	7.01	99.58	30.09

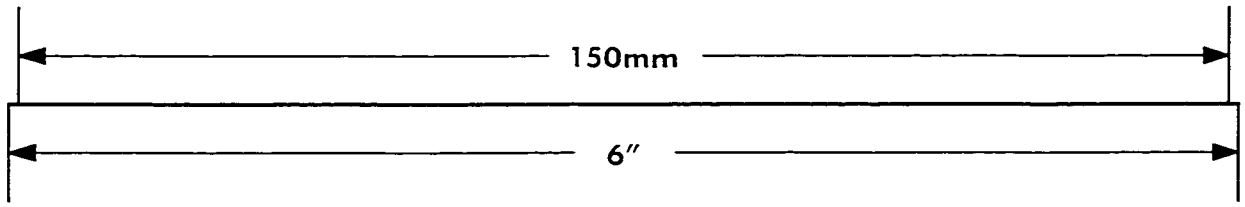
Table 6: *Coverage achieved (in %) and time required to attain that coverage for all edges and for unconstrained edges, given for all analyzed programs*

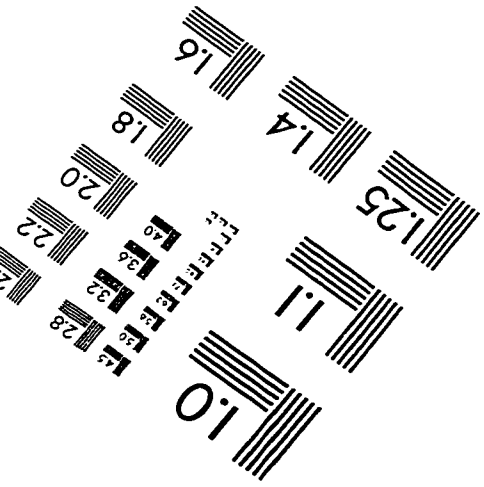
IMAGE EVALUATION
TEST TARGET (QA-3)

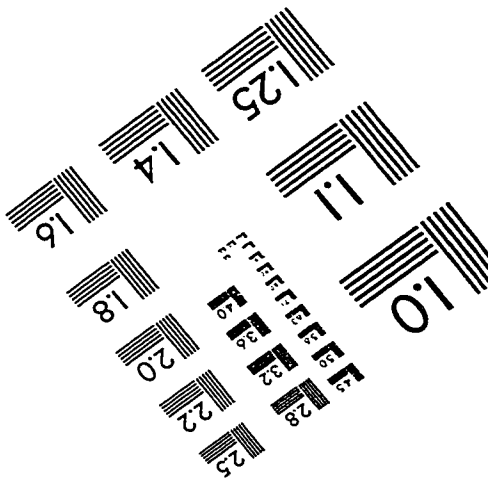












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989